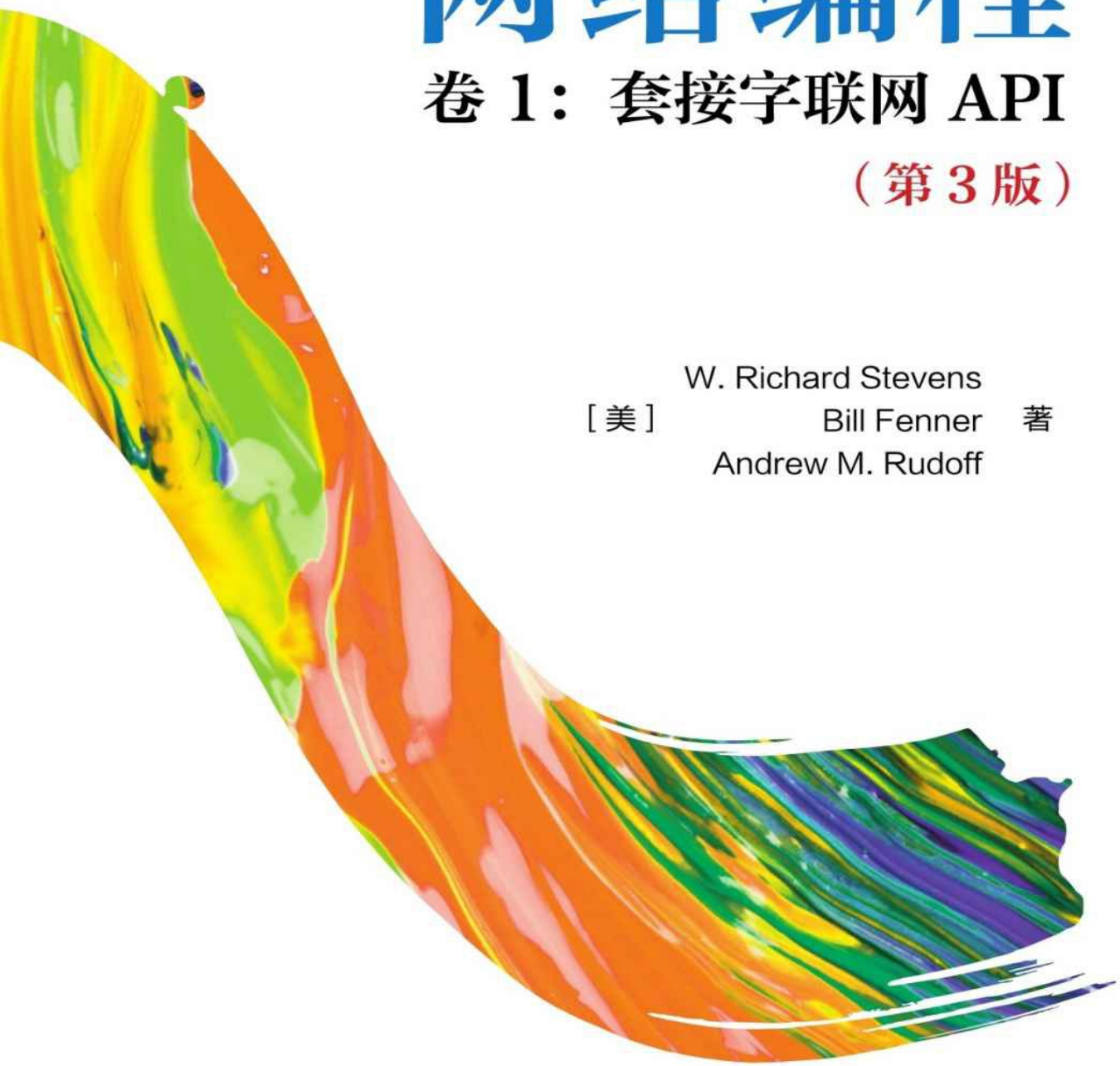
PEARSON

UNIX 网络编程

卷 1：套接字联网 API

（第 3 版）

[美] W. Richard Stevens
Bill Fenner
Andrew M. Rudoff 著



中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

Unix Network Programming, Volume 1: The Sockets Networking API Third Edition

目　录

PEARSON

UNIX网络编程　卷1：套接字联网API（第3版）

[美]W.Richard Stevens　Bill Fenner　Andrew M.Rudoff　著

人民邮电出版社

北京

图书在版编目（**CIP**）数据

UNIX网络编程：第3版.第1卷，套接字联网API/（美）史蒂文斯（Stevens,W.R.），（美）芬纳（Fenner,B.），（美）鲁道夫（Rudoff,A.M.）著；匿名译.--2版.--北京：人民邮电出版社，2015.8

书名原文：Unix Network Programming，Volume 1:The Sockets Networking API,Third Edition

ISBN 978-7-115-36719-8

Ⅰ.①U… Ⅱ.①史…②芬…③鲁…④匿… Ⅲ.①UNIX操作系统—程序设计 Ⅳ.①TP316.81

中国版本图书馆CIP数据核字（2015）第143512号

内容提要

本书是一部UNIX网络编程的经典之作！书中全面深入地介绍了如何使用套接字API进行网络编程。全书不但介绍了基本编程内容，还涵盖了与套接字编程相关的高级主题，对于客户/服务器程序的各种设计方法也作了完整的探讨，最后还深入分析了流这种设备驱动机制。

本书内容详尽且具权威性，几乎每章都提供精选的习题，并提供了部分习题的答案，是网络研究和开发人员理想的参考书。

◆著 [美]W.Richard Stevens Bill Fenner Andrew M.Rudoff

责任编辑 杨海玲

责任印制 张佳莹 焦志炜

◆人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 http://www.ptpress.com.cn

北京艺辉印刷有限公司印刷

◆开本：787×1092 1/16

印张：51.5

字数：1363千字 2015年8月第2版

印数：1-4500册　　2015年8月北京第1次印刷
著作权合同登记号　图字：01-2009-5715号
定价：129.00元
读者服务热线：**(010)81055410**　印装质量热线：**(010)81055316**
反盗版热线：**(010)81055315**
广告经营许可证：京崇工商广字第**0021**号

版权声明

序

本书的第1版于1990年问世，并迅速成为程序员学习网络编程的权威参考书。时至今日，计算机网络技术已发生了翻天覆地的变化。只要看看第1版给出的用于征集反馈意见的地址（“uunet!hsi!netbook”）就一目了然了。（有多少读者能看出这是20世纪80年代很流行的UUCP拨号网络的地址？）

现在UUCP网络已经很罕见了，而无线网络等新技术则变得无处不在！在这种背景下，新的网络协议和编程范型业已开发出来，但程序员却苦于找不到一本好的参考书来学习这些复杂的新技术。

这本书填补了这一空白。拥有本书旧版的读者一定想要一个新的版本来学习新的编程方法，了解IPv6等下一代协议方面的新内容。所有人都非常期待本书，因为它完美地结合了实践经验、历史视角以及在本领域浸淫多年才能获得的透彻理解。

阅读本书是一种享受，我收获颇丰。相信大家定会有同感。

Sam Leffler

前言

概述

本书面向的读者是那些希望自己编写的程序能使用称为套接字（socket）的API进行彼此通信的人。有些读者可能已经非常熟悉套接字了，因为这个模型几乎已经成了网络编程的同义词，但有些读者可能仍需要从头开始学习。本书想达到的目标是向大家提供网络编程指导。这些内容不仅适用于专业人士，也适用于初学者；不仅适用于维护已有代码，也适用于开发新的网络应用程序；此外，还适用于那些只是想了解一下自己系统中网络组件的工作原理的人。

书中的所有示例都是在Unix系统上测试通过的真实的、可运行的代码。但是，考虑到许多非Unix的操作系统也支持套接字API，因而我们选取的示例与所讲述的一般性概念，在很大程度上是与操作系统无关的。几乎每种操作系统都提供了大量的网络应用程序，如网页浏览器、电子邮件客户端、文件共享服务器等。我们按常规的划分方法把这些应用程序分为客户程序和服务器程序，并在书中多次编写了相应的小型示例。

面向Unix介绍网络编程自然免不了要介绍Unix本身和TCP/IP的相关背景知识。需要更详尽的背景知识时，我们会指引读者查阅其他书籍。本书中经常提到以下4本书，我们将其简记如下。

APUE：Advanced Programming in the UNIX Environment [Stevens 1992]。

TCPv1：TCP/IP Illustrated, Volume 1 [Stevens 1994]。

TCPv2：TCP/IP Illustrated, Volume 2 [Wright and Stevens 1995]。

TCPv3：TCP/IP Illustrated, Volume 3 [Stevens 1996]。

其中TCPv2包含了与本书内容密切相关的细节，它描述并给出了套接字API中网络编程函数（socket、bind、connect等）的真实4.4BSD实现。如果已经理解某个特性的实现，那么在应用程序中使用该特性就更有意义了。

与第2版的区别

从20世纪80年代开始，套接字就差不多是现在这个样子了。时至今日，套接字仍然是网络API的首选，其最初的设计的确值得称道。因此，当读者发现我们对出版于1998年的第2版又做了不少改动时，可能会觉得惊讶。本书中所做的改动归纳如下。

新版本包含了IPv6的最新信息。在第2版出版时，IPv6尚处于草案阶段，这些年来已经有所发展。

更新了全部函数和示例的描述，以反映最新的POSIX规范（POSIX 1003.1-2001），即Single Unix Specification Version 3。

删去了X/Open传输接口（XTI）的内容。这个API已经不常用了，连最新的POSIX 规范也不再提到。

删去了事务TCP协议（T/TCP）的内容。

新增了三章用于描述一种相对较新的传输协议——SCTP。这个可靠的面向消息的协议能够在两个端点之间提供多个流，并为多归属技术提供传输层支持。该协议最初是为了在因特网上传输电话信号而设计的，但它的一些特性可以用于许多应用。

新增一章描述密钥管理套接字，该套接字可用于网际协议安全（IPsec）和其他网络安全服务。

第2版中使用的机器及Unix变体都按最新版本更新，示例也根据机器的特性做了修改。许多情况下，修改示例是因为操作系统厂商修正了程序缺陷或者新增了特性。但读者可以想见，新的缺陷总能不时地被发

现。本书中用于测试示例的机器如下：

运行MacOS/X 10.2.6的Apple Power PC；

运行HP-UX 11i 的HP PA-RISC；

运行AIX 5.1的IBM Power PC；

运行FreeBSD 4.8的Intel x86；

运行Linux 2.4.7的Intel x86；

运行FreeBSD 5.1的Sun SPARC；

运行Solaris 9的Sun SPARC。

这些机器的具体用法见图1-16。

本系列的第2卷（《UNIX网络编程　卷2：进程间通信》）基于本卷的内容进一步讨论了消息传递、同步、共享内存及远程过程调用。

如何使用本书

本书既可以作为网络编程的教程，也可以作为有经验的程序员的参考书。用作网络编程的教程或入门级教材时，重点应放在第二部分（第3章至第11章），然后可以看看其他感兴趣的主题。第二部分包含了TCP和UDP的基本套接字函数，以及SCTP、I/O多路复用、套接字选项和基本名字与地址的转换。所有读者都应该阅读第1章，尤其是1.4节，介绍了一些贯穿全书的包裹函数。读者可以根据自身的知识背景，选读第2章，或许还有附录A。第三部分的多数章节可以彼此独立地进行阅读。

为了方便读者把本书作为参考书，本书提供了完整的全文索引，并在最后几页总结了每个函数和结构的详细描述在正文中的哪里可以找到。为了给不按顺序阅读本书的读者提供方便，我们在全书中为相关主题提供了大量的交叉引用。

源代码与勘误

书中所有示例的源代码可以从www.unpbook.com获得 [1] 。学习网络编程的最好方法就是下载这些程序，对其进行修改和改进。只有这样

实际编写代码才能深入理解有关概念和方法。每章末尾提供了大量的习题，大部分在附录E中给出答案。

本书的最新勘误表也可以在上述网站获取。

致谢

本书第1版和第2版由W. Richard Stevens独立撰写，他不幸于1999年9月1日去世。Richard的著作体现了非常高的水准，被公认为是精练、翔实且极具可读性的艺术作品。在撰写这一修订版的过程中，我们力图保持Richard之前版本的高质量和全面性，这方面的任何不足都完全是新作者的过错。

任何作者的著作离不开家人与朋友的支持。Bill Fenner在此感谢爱妻Peggy（沙滩1/4英里赛冠军）与好友Christopher Boyd在本书撰写过程中承担了全部的家务，还要感谢朋友Jerry Winner，他的激励是无价的。同样地，Andy Rudoff要特别感谢他的妻子Ellen和两个女儿Jo、Katie自始至终的理解与鼓励。没有你们的支持，我们不可能完成本书。

思科公司的Randall Stewart提供了许多SCTP的材料，非常感谢他的巨大贡献。如果缺少了他的工作，本书就不能涵盖这一新颖而有趣的主题。

本书的审稿人给出了宝贵的反馈意见。他们发现了一些错误，指出了一些需要更多解释的地方，并对文字和代码示例提出了一些改进建议。作者在这里对如下审稿人表示感谢：James Carlson、Wu-Chang Feng、Rick Jones、Brian Kernighan、Sam Leffler、John McCann、Craig Metz、Ian Lance Taylor、David Schwartz和Gary Wright。

许多个人及其单位为本书中一些示例的测试提供了帮助，他们义务向我们出借系统、软件或为我们提供系统访问权限。

IBM奥斯汀实验室的Jessie Haug提供了AIX系统和编译器。

惠普公司的Rick Jones和William Gilliam为我们提供了运行HP-UX的多个系统的访问权限。

与Addison Wesley出版社的员工合作非常愉快，他们是Noreen Regina、Kathleen Caren、Dan DePasquale和Anthony Gemellaro。要特别感谢本书的编辑Mary Franz。

为了延续Rich Stevens的风格（不过该风格与流行的风格相反），我们用James Clark编写的优秀的Groff包为本书排版，用gpic程序绘制插图（其中用到了许多由Gary Wright编写的宏），用gtbl程序生成了表格，我们为全书添加了索引，并设计了最终的版式。录入源代码时用到了Dave Hanson的loom程序和Gary Wright写的一些脚本。在生成最终索引的过程中，还用到了Jon Bentley与Brian Kernighan编写的一组awk脚本。

欢迎读者以电子邮件的方式反馈意见、提出建议或订正错误。

Bill Fenner

加利福尼亚州伍德赛德市

Andrew M. Rudoff

科罗拉多州博尔德市

2003年10月

authors@unpbook.com

http://www.unpbook.com

---

[1]. 书中所有示例的源代码也可以从图灵网站（www.turingbook.com）本书网页免费注册下载。——编者注

第一部分 简介和TCP/IP

第1章 简介

1.1 概述

要编写通过计算机网络通信的程序，首先要确定这些程序相互通信所用的协议（protocol）。在深入设计一个协议的细节之前，应该从高层次决断通信由哪个程序发起以及响应在何时产生。举例来说，一般认为Web服务器程序是一个长时间运行的程序（即所谓的守护程序，daemon），它只在响应来自网络的请求时才发送网络消息。协议的另一端是Web客户程序，如某种浏览器，与服务器进程的通信总是由客户进程发起。大多数网络应用就是按照划分成客户（client）和服务器（server） [1] 来组织的。在设计网络应用 [2] 时，确定总是由客户发起请求往往能够简化协议和程序 [3] 本身。当然一些较为复杂的网络应用还需要异步回调（asynchronous callback）通信，也就是由服务器向客户发起请求消息。然而坚持采纳图1-1所示的基本客户/服务器模型的网络应用毕竟要普遍得多。

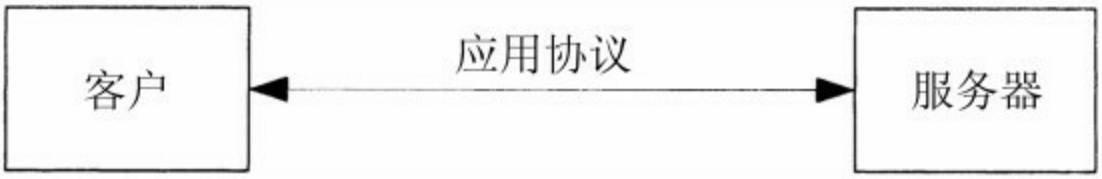


图1-1 网络应用：客户和服务器

通常客户每次只与一个服务器通信，不过以使用Web浏览器为例，我们也许在10分钟内就可以与许多不同的Web服务器通信。从服务器的角度来看，一个服务器同时与多个客户通信并不稀奇，见图1-2。本书后面将介绍若干种让一个服务器同时处理多个客户请求的方法。

可认为客户与服务器之间是通过某个网络协议通信的，但实际上，这样的通信通常涉及多个网络协议层。本书的焦点是TCP/IP协议族，也称为网际协议族。举例来说，Web客户与服务器之间使用TCP（Transmission Control Protocol，传输控制协议）通信。TCP又转而使用IP（Internet Protocol，网际协议）通信，IP再通过某种形式的数据链路层通信。如果客户与服务器处于同一个以太网，就有图1-3所示的通信层次。

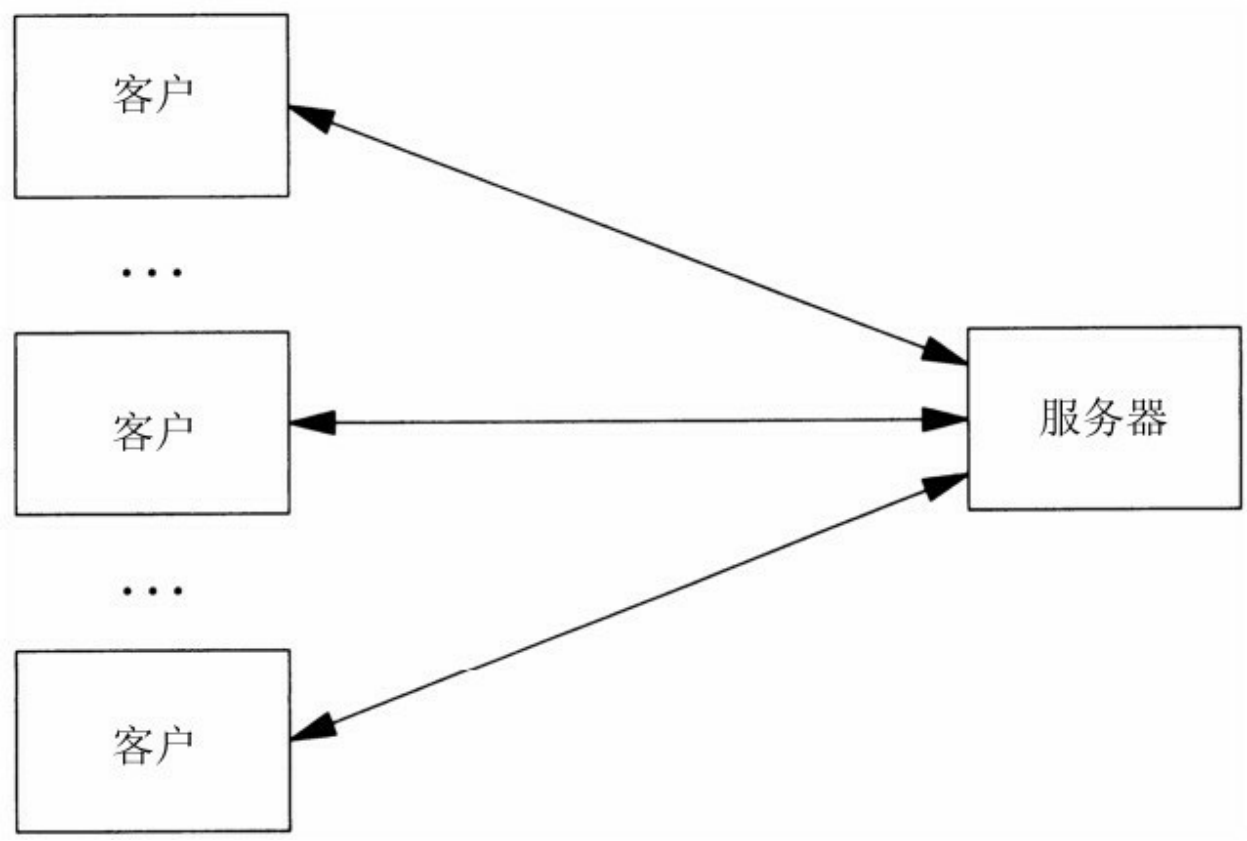


图1-2 一个服务器同时处理多个客户的请求

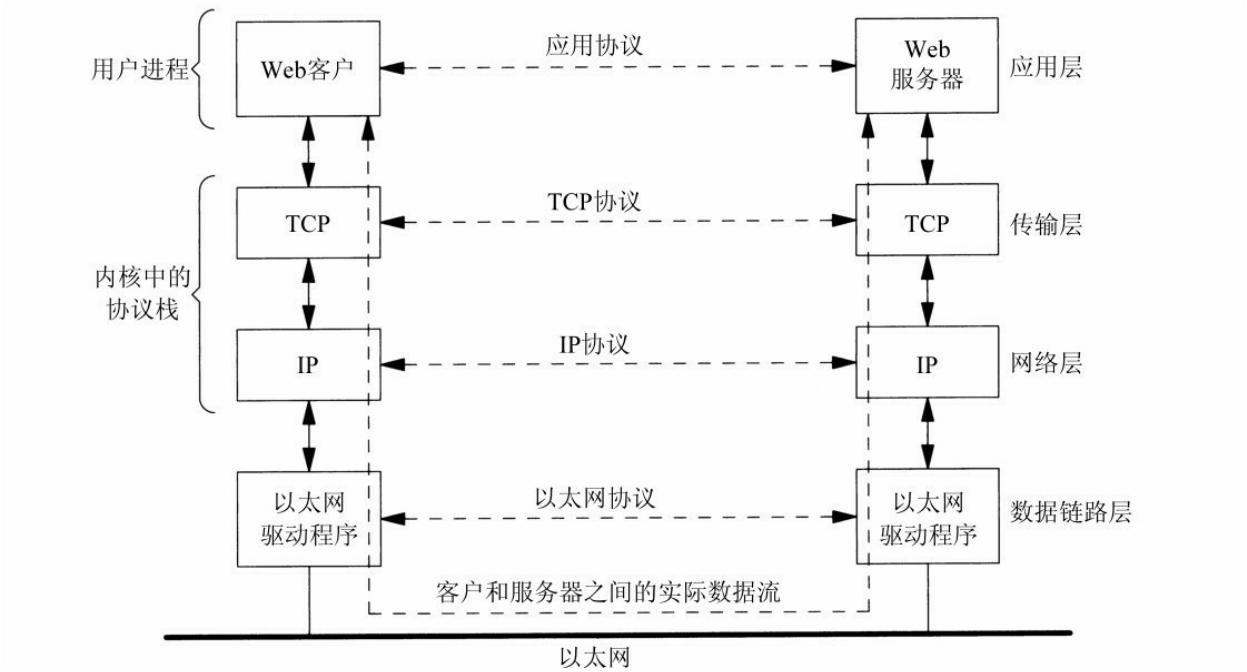


图1-3 客户与服务器使用TCP在同一个以太网中通信

尽管客户与服务器之间使用某个应用协议通信，传输层却使用TCP通信。注意，客户与服务器之间的信息流在其中一端是向下通过协议栈的，跨越网络后，在另一端则是向上通过协议栈的。另外注意，客户和服务器通常是用户进程，而TCP和IP协议通常是内核中协议栈的一部分。我们在图1-3右边标出了4个层。

本书讨论的协议不限于TCP和IP。有些客户和服务器改用UDP（User Datagram Protocol，用户数据报协议）而不是TCP，第2章将详细介绍这两个协议。此外，本书使用术语“IP”来称谓的那个协议，自20世纪80年代早期以来一直在使用，其实其正式名称是IPv4（IP version 4，IP版本4）。IPv4的一个新版本IPv6（IP version 6，IP版本6）是在20世纪90年代中期开发出来的，将来会取代IPv4。本书既讨论使用IPv4的网络应用程序的开发，也讨论使用IPv6的网络应用程序的开发。附录A会给出IPv4和IPv6的一个比较，同时介绍正文中将讨论的其他协议。

同一网络应用的客户和服务器无需如图1-3所示处于同一个局域网（local area network， LAN）。例如，图1-4展示了处于不同局域网中的

客户和服务器，而这两个局域网是使用路由器（router）连接到广域网（wide area network，WAN）的。

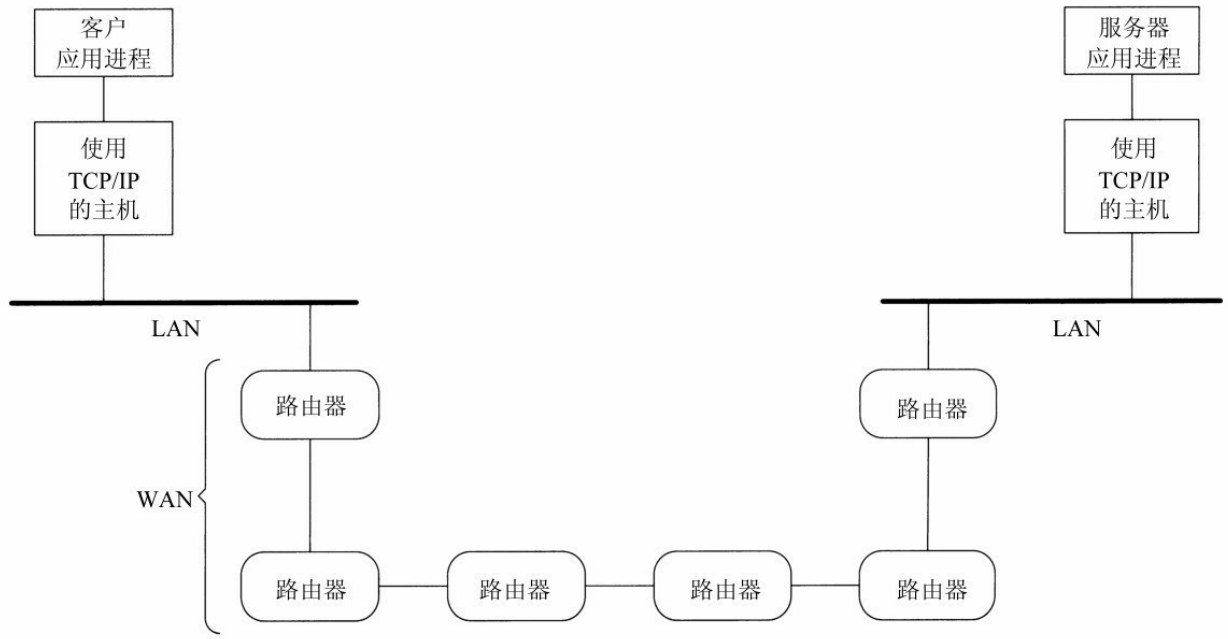


图1-4 处于不同局域网的客户主机和服务器主机通过广域网连接

路由器是广域网的架构设备。当今最大的广域网是因特网[4]（Internet）。许多公司也构建自己的广域网，而这些私用的广域网既可以连接到因特网，也可以不连接到因特网。

本章其余部分将概述多个主题，这些主题在后续章节中还会具体介绍。我们从一个尽管简单却完整的TCP客户程序开始，它展示了全书都会遇到的许多函数调用和概念。这个客户程序只能在IPv4上运行，不过我们会给出让它在IPv6上运行所需进行的修改。更好的办法是编写独立于协议的客户和服务器程序，这在第11章中会讨论。本章同时展示一个与该TCP客户程序配合工作的完整的TCP服务器程序。

为了简化代码，我们对本书中要调用的大多数系统函数定义了各自的包裹函数。多数情况下我们可以使用这些包裹函数来检查错误，输出适当的消息，以及在出错时终止程序的运行。我们还给出了本书中大多数例子所用的测试网络、主机、路由器以及它们的主机名、IP地址和操

作系统。

如今讨论Unix时经常使用POSIX一词，它是一种被多数厂商采纳的标准。我们将介绍POSIX的历史以及它对本书所讲述的API的影响，并介绍该领域的其他主要标准。

1.2 一个简单的时间获取客户程序

让我们考虑一个具体的例子，引入将在本书中遇到的许多概念和说法。图1-5所示的是TCP当前时间查询客户程序的一个实现。该客户与其服务器建立一个TCP连接后，服务器以直观可读格式简单地送回当前时间和日期。

intro/daytimetcpcli.c

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd, n;
 6     char    recvline[MAXLINE + 1];
 7     struct sockaddr_in  servaddr;
 8     if (argc != 2)
 9         err_quit("usage: a.out <IPaddress>");
10     if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");
12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin_family = AF_INET;
14     servaddr.sin_port   = htons(13);   /* daytime server */
15     if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
16         err_quit("inet_pton error for %s", argv[1]);
17     if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
18         err_sys("connect error");
19     while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
20         recvline[n] = 0;        /* null terminate */
21         if (fputs(recvline, stdout) == EOF)
22             err_sys("fputs error");
23     }
24     if (n < 0)
25         err_sys("read error");
26     exit(0);
27 }
```

intro/daytimetcpcli.c

图1-5 TCP时间获取客户程序

这就是本书用于展示所有源代码的格式。每个非空行都被编排行号。如稍后所示，代码正文讲解部分一开始标注该段代码起始与结束的行号。有的段落会以一个简短的、描述性的醒目标题起头，对所讲解代码段进行概要说明。

每个源代码段起始与结束处的水平线标出了该代码段所在的源代码文件名，对于本例就是intro目录下的daytimetcpcli.c文件（intro/daytimetcpcli.c）。本书所有例子的源代码都可免费获得（见前言），在此标注它们的文件名便于读者找到其源文件。在阅读本书期

间，编译、运行特别是修改这些程序是学习网络编程概念的好方法。

整本书中我们随时会插入缩进的小字号段落（如此处所示）来说明实现的细节和历史上的观点。

如果编译该程序生成默认的a.out可执行文件后执行它，我们会得到如下结果：

solaris % a.out 206.168.112.96　　我们的输入

Mon May 26 20:58:40 2003　　程序的输出

当我们展示交互的输入和输出时，输入总是采用加粗的等宽字体，而计算机的输出总是采用不加粗的等宽字体。注释用宋体字加在右边。作为shell提示一部分的系统名字（本例中为solaris）指明在哪个主机上执行该命令。图1-16展示了用于运行本书中大多数例子的各个系统，它们的主机名本身通常就说明了各自的操作系统。

在这个短短27行的程序中有许多细节值得考虑。这里我们简短地提一下，目的是让初次遇到网络程序的读者有所准备，本书后面会更详细地说明这些内容。

包含头文件

1　包含我们自己编写的名为unp.h的头文件，见D.1节。该头文件包含了大部分网络程序都需要的许多系统头文件，并定义了所用到的各种常值 [5] （如MAXLINE）。

命令行参数

2～3　这是main函数的定义，其形式参数就是命令行参数。本书中的代码假设使用ANSI C编译器（也称为ISO C编译器）编写。

创建**TCP**套接字

10～11　　socket函数创建一个网际（AF_INET）字节流（SOCK_STREAM）套接字，它是TCP套接字的花哨名字。该函数返回一个小整数描述符，以后的所有函数调用（如随后的connect和read）就

用该描述符来标识这个套接字。

if语句包含3个操作：调用socket函数，把返回值赋给变量sockfd，再测试所赋的这个值是否小于0。虽然我们可以把该语句分割成两条C语句：

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
```

但是把这两行合并成一行却是常见的C语言习惯用法。按照C语言的优先规则（小于运算符的优先级高于赋值运算符），函数调用和赋值语句外边的那对括号是必需的。作为一种编码风格，作者总是在这样的两个左括号间加一个空格，提示比较运算的左侧同时也是一个赋值运算。（这种风格借鉴自Minix源代码［Tenenbaum 1987］。）该程序稍后的while语句也使用相同的样式。

后面我们将遇到术语套接字（socket [6] ）的许多不同用法。首先，我们正在使用的API称为套接字API（sockets API）。上一段中名为socket的函数就是套接字API的一部分。上一段中我们还提到了“TCP套接字”，它是“TCP端点”（TCP endpoint）的同义词。如果socket函数调用失败，我们就调用自己的err_sys函数放弃程序运行。err_sys函数输出我们作为参数提供的出错消息以及所发生的系统错误的描述（例如出自socket函数的可能错误之一“Protocol not supported”（协议不受支持）），然后终止进程。这个函数和以err_开头的其他若干个函数都是我们自行编写的，它们的调用将贯穿全书，D.3节会描述这些函数。

指定服务器的**IP**地址和端口

12～16 我们把服务器的IP地址和端口号填入一个网际套接字地址结构（一个名为servaddr的sockaddr_in结构变量）。使用bzero把整个结构清零后，置地址族为AF_INET，端口号为13（这是时间获取服务器的众所周知端口，支持该服务的任何TCP/IP主机都使用这个端口号，见图2-18），IP地址为第一个命令行参数的值（argv[1]）。网际套接字地址结

构中IP地址和端口号这两个成员必须使用特定格式，为此我们调用库函数htons（“主机到网络短整数”）去转换二进制端口号，又调用库函数inet_pton（“呈现形式到数值”）去把ASCII命令行参数（例如运行本例子所用的206.168.112.96）转换为合适的格式。

bzero不是一个ANSI　C函数。它起源于早期的Berkeley网络编程代码。不过我们在整本书中使用它而不用ANSI　　C的memset函数，因为bzero（带2个参数）比memset（带3个参数）更好记忆。几乎所有支持套接字API的厂商都提供bzero，如果没有，那么可以使用unp.h头文件中提供的该函数的宏定义。

事实上，在TCPv3一书首次印刷时，作者在10处出现memset函数的地方犯了错，互换了第二和第三个参数。C编译器发现不了这个错误，因为这两个参数的类型是相同的。（其实第二个参数是int类型，第三个参数是size_t，通常定义为unsigned　int类型，然而分别指定给这两个参数的值为0和16，它们对于两个参数的类型同样可以接受。）对memset的这些调用仍然正常，不过没做任何事，因为待初始化的字节数被指定成了0。程序之所以仍然工作是因为只有少数套接字函数要求网际套接字地址结构的最后8个字节置0。无论如何，这确实是一个错误，且是一个通过使用bzero函数可以避免的错误，因为如果使用函数原型，C编译器总能发现bzero的两个参数被互换的错误。

此处也许是你第一次遇到inet_pton函数。它是一个支持IPv6（详见附录A）的新函数。以前的代码使用inet_addr函数来把ASCII点分十进制数串变换成正确的格式，不过它有不少局限，而这些局限在inet_pton中都得以纠正。如果你的系统尚未支持该函数，那你可以使用我们在3.7节中提供的它的一个实现。

建立与服务器的连接

17～18 connect函数应用于一个TCP套接字时，将与由它的第二个参数指向的套接字地址结构指定的服务器建立一个TCP连接。该套接字地

址结构的长度也必须作为该函数的第三个参数指定，对于网际套接字地址结构，我们总是使用C语言的sizeof操作符由编译器来计算这个长度。

在头文件unp.h中，我们使用#define把SA定义为struct sockaddr，也就是通用套接字地址结构。每当一个套接字函数需要一个指向某个套接字地址结构的指针时，这个指针必须强制类型转换成一个指向通用套接字地址结构的指针。这是因为套接字函数早于ANSI C标准，20世纪80年代早期开发这些函数时，ANSI C的void *指针类型还不可用。问题是“struct sockaddr”长达15个字符，往往造成源代码行超出屏幕（或者书页，若是排印在书上）的右边缘，因此我们把它缩减成SA。我们将在解释图3-3时详细讨论通用套接字地址结构。

读入并输出服务器的应答

19～25 我们使用read函数读取服务器的应答，并用标准的I/O函数fputs输出结果。 [7] 使用TCP时必须小心，因为TCP是一个没有记录边界的字节流协议。服务器的应答通常是如下格式的26字节字符串：

Mon May 26 20:58:40 2003\r\n

其中，\r是ASCII回车符，\n是ASCII换行符。使用字节流协议的情况下，这26个字节可以有多种返回方式：既可以是包含所有26个字节的单个TCP分节 [8] ，也可以是每个分节只含1个字节的26个TCP分节，还可以是总共26个字节的任何其他组合。通常服务器返回包含所有26个字节的单个分节，但是如果数据量很大，我们就不能确保一次read调用能返回服务器的整个应答。因此从TCP套接字读取数据时，我们总是需要把read编写在某个循环中，当read返回0（表明对端关闭连接）或负值（表明发生错误）时终止循环。

本例中，服务器关闭连接表征记录的结束。HTTP（Hypertext Transfer Protocol，超文本传送协议）的1.0版本也采用这种技术。还可以用其他技术标记记录结束。例如，SMTP （Simple Mail Transfer Protocol，简单邮件传送协议）使用由ASCII回车符后跟换行符构成的2

字节序列标记记录的结束；Sun远程过程调用（Remote Procedure Call，RPC）以及域名系统（Domain Name System，DNS）在使用TCP承载应用数据时，在每个要发送的记录之前放置一个二进制的计数值，给出这个记录的长度。这里的重要概念是TCP本身并不提供记录结束标志：如果应用程序需要确定记录的边界，它就要自己去实现，已有一些常用的方法可供选择。

终止程序

26　exit终止程序运行。Unix在一个进程终止时总是关闭该进程所有打开的描述符，我们的TCP套接字就此被关闭。

刚才已提过，本书后面会对刚才讲述的所有概念深入进行探讨。

1.3 协议无关性

图1-5中的程序是与IPv4协议相关的：我们分配并初始化一个sockaddr_in类型的结构，把该结构的协议族成员设置为AF_INET，并指定socket函数的第一个参数为AF_INET。 ■

为了让图1-5中的程序能够在IPv6上运行，我们必须修改这段代码。图1-6所示的是一个能够在IPv6上运行的版本，其中改动之处用加粗的等宽字体突出显示。

intro/daytimetcpcliv6.c

```
 1 #include    "unp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd, n;
 6     char    recvline[MAXLINE + 1];
 7     struct sockaddr_in6 servaddr;
 8     if (argc != 2)
 9         err_quit("usage: a.out <IPaddress>");
10     if ( (sockfd = socket(AF_INET6, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");
12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin6_family = AF_INET6;
14     servaddr.sin6_port   = htons(13);  /* daytime server */
15     if (inet_pton(AF_INET6, argv[1], &servaddr.sin6_addr) <= 0)
16         err_quit("inet_pton error for %s", argv[1]);
17     if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
18         err_sys("connect error");
19     while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
20         recvline[n] = 0;    /* null terminate */
21         if (fputs(recvline, stdout) == EOF)
22             err_sys("fputs error");
23     }
24     if (n < 0)
25         err_sys("read error");
26     exit(0);
27 }
```

intro/daytimetcpcliv6.c

图1-6 适合于IPv6的图1-5所示程序的修改版

我们只修改了程序的5行代码，得到的却是另一个与协议相关的程序：这回是与IPv6协议相关的。更好的做法是编写协议无关的程序。图11-11将给出本客户程序的协议无关版本，它使用了getaddrinfo函数（由tcp_connect函数调用）。

这两个程序的另一个不足之处是：用户必须以点分十进制数格式给出服务器的IP地址（如适合于IPv4版本的206.168.112.219）。人们更习惯于用名字（如www.unpbook.com）来代替数字。我们将在第11章中讨论主机名与IP地址之间以及服务名与端口之间的转换函数。我们特意推

迟讨论这些函数，在第11章之前继续使用IP地址和端口号，目的是了解我们必须填写和查看的套接字地址结构的细节，避免被另一个函数集的细节把网络编程的讨论搞复杂了。

1.4 错误处理：包裹函数

任何现实世界的程序都必须检查每个函数调用是否返回错误。在图1-5所示的程序中，我们检查socket、inet_pton、connect、read和fputs函数是否返回错误，当发生错误时，就调用我们自己的err_quit或err_sys函数输出一个出错消息并终止程序的运行。我们发现绝大多数情况下这正是我们想做的事。个别情况下，当这些函数返回错误时，我们想做的事并非简单地终止程序的运行，如图5-12所示，我们必须检查系统调用是否被中断了。

既然发生错误时终止程序的运行是普遍的情况，我们可以通过定义包裹函数（wrapper function）来缩短程序。每个包裹函数完成实际的函数调用，检查返回值，并在发生错误时终止进程。我们约定包裹函数名是实际函数名的首字母大写形式。例如，在语句

sockfd = Socket(AF_INET, SOCK_STREAM, 0);

中，函数Socket是函数socket的包裹函数，如图1-7所示。

lib/wrapsock.c

```
236 int
237 Socket(int family, int type, int protocol)
238 {
239     int     n;

240     if ( (n = socket(family, type, protocol)) < 0)
241         err_sys("socket error");
242     return(n);
243 }
```

lib/wrapsock.c

图1-7 socket函数的包裹函数

在本书中只要你遇到一个首字母大写的函数名，它就是我们定义的某个包裹函数。它调用的实际函数的名字与包裹函数名相同，不过以对

应的小写字母开头。

然而在讲解本书中提供的源代码时，我们总是指称被调用的最低级别的函数（如socket），而不是包裹函数（如Socket）。

这些包裹函数不见得多节省代码量，但当我们在第26章中讨论线程时，将会发现线程函数遇到错误时并不设置标准Unix的errno变量，而是把errno的值作为函数返回值返回调用者。这意味着每次调用以pthread_开头的某个函数时，我们必须分配一个变量来存放函数返回值，以便在调用err_sys前把errno变量设置成该值。为避免引入花括号把代码弄得很混乱，我们可以使用C语言的逗号操作符，把errno的赋值与err_sys的调用组合成一条语句，如下所示：

```
int        n;
if ( (n = pthread_mutex_lock(&ndone_mutex)) != 0)
   errno = n, err_sys("pthread_mutex_lock error");
```

我们也可以为此定义一个新的错误处理函数，它取系统的错误号作为一个参数，不过通过定义如图1-8所示的包裹函数，我们可以让以上这段代码更为易读：

```
Pthread_mutex_lock(&ndone_mutex);
```

要是仔细推敲C代码的编写，我们可以用宏来替代函数，从而稍微提高运行时效率，不过包裹函数很少是程序性能的瓶颈所在。

选择首字母大写一个函数名作为其包裹函数名是一种折中的方法。其他方法也考虑过，譬如给函数名加一个“e”前缀（如［Kernighan and Pike 1984］一书第182页所示），给函数名加一个“_e”后缀，等等。这些方法都能明显地提示调用了其他函数，但我们的这种风格看来是最少分散注意力的。

这种技术还有助于检查那些错误返回值通常被忽略的函数是否出错，例如close和listen。

lib/wrappthread.c

```
72 void
73 Pthread_mutex_lock(pthread_mutex_t *mptr)
74 {
75     int     n;

76     if ( (n = pthread_mutex_lock(mptr)) == 0)
77         return;
78     errno = n;
79     err_sys("pthread_mutex_lock error");
80 }
```

lib/wrappthread.c

图1-8 pthread_mutex_lock的包裹函数

本书后面的例子中，除非必须检查某个确定的错误是否发生，并以不同于终止进程的其他某种方式处理它，否则就使用这些包裹函数。书中不提供所有包裹函数的源代码，不过它们是可以免费获得的（见前言）。

Unix errno值

只要一个Unix函数（例如某个套接字函数）中有错误发生，全局变量errno就被置为一个指明该错误类型的正值，函数本身则通常返回-1。err_sys查看errno变量的值并输出相应的出错消息，例如当errno值等于ETIMEDOUT时，将输出“Connection timed out”（连接超时）。

errno的值只在函数发生错误时设置。如果函数不返回错误，errno的值就没有定义。errno的所有正数错误值都是常值，具有以“E”开头的全大写字母名字，并通常在<sys/errno.h>头文件中定义。值0不表示任何错误。

在全局变量中存放errno值对于共享所有全局变量的多个线程并不适合。我们将在第26章中讲述解决这一问题的方法。

全书中我们将使用诸如“connect函数返回ECONNREFUSED”这样的句子简明表达以下意思：该函数返回一个错误（通常函数返回值为-1），同时errno被置为指定的常值。

1.5 一个简单的时间获取服务器程序

我们可以编写一个简单的TCP时间获取服务器程序，它和1.2节中的客户程序一道工作。图1-9给出了这个服务器程序，它使用了上一节中讲过的包裹函数。

创建**TCP**套接字

10 TCP套接字的创建与客户程序相同。

把服务器的众所周知端口捆绑到套接字

11～15 通过填写一个网际套接字地址结构并调用bind函数，服务器的众所周知端口（对于时间获取服务是13）被捆绑到所创建的套接字。我们指定IP地址为INADDR_ANY，这样要是服务器主机有多个网络接口，服务器进程就可以在任意网络接口上接受客户连接。

以后我们将了解怎样限定服务器进程只在单个网络接口上接受客户连接。

intro/daytimetcpsrv.c

```
 1 #include    "unp.h"
 2 #include    <time.h>
 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     struct sockaddr_in  servaddr;
 8     char    buff[MAXLINE];
 9     time_t  ticks;
10     listenfd = Socket(AF_INET, SOCK_STREAM, 0);
11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14     servaddr.sin_port = htons(13);  /* daytime server */
15     Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
16     Listen(listenfd, LISTENQ);
17     for ( ; ; ) {
18         connfd = Accept(listenfd, (SA *) NULL, NULL);
19         ticks = time(NULL);
20         snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
21         Write(connfd, buff, strlen(buff));
22         Close(connfd);
23     }
24 }
```

intro/daytimetcpsrv.c

图1-9 TCP时间获取服务器程序

把套接字转换成监听套接字

16　调用listen函数把该套接字转换成一个监听套接字，这样来自客户的外来连接就可在该套接字上由内核接受。socket、bind和listen这3个调用步骤是任何TCP服务器准备所谓的监听描述符（listening descriptor，本例中为listenfd）的正常步骤。

常值LISTENQ在我们的unp.h头文件中定义。它指定系统内核允许在这个监听描述符上排队的最大客户连接数。我们将在4.5节详细说明客户连接的排队。

接受客户连接，发送应答

17～21　通常情况下，服务器进程在accept调用中被投入睡眠，等待

某个客户连接的到达并被内核接受。TCP连接使用所谓的三路握手（three-way handshake）来建立连接。握手完毕时accept返回，其返回值是一个称为已连接描述符（connected descriptor）的新描述符（本例中为connfd）。该描述符用于与新近连接的那个客户通信。accept为每个连接到本服务器的客户返回一个新描述符。

本书全文采用的无限循环采用以下风格：

```
for ( ; ; ) {
   . . .
}
```

当前时间和日期是由库函数time返回的，它实际上返回的是自Unix纪元即1970年1月1日0点0分0秒（国际标准时间）以来的秒数。下一个库函数ctime把该整数值转换成直观可读的时间格式，例如：

Mon May 26 20:58:40 2003

snprintf函数在这个字符串末尾添加一个回车符和一个回行符，随后write函数把结果字符串写给客户。

如果你尚不习惯改用snprintf代替较早的sprintf函数，那么现在是学习的时候了。调用sprintf无法检查目的缓冲区是否溢出。相反，snprintf要求其第二个参数指定目的缓冲区的大小，因此可确保该缓冲区不溢出。

snprintf相对较晚才加到ANSI C标准中，在称为ISO C99的版本中引入。不过几乎所有厂商都把它作为标准C函数库的一部分提供，而且另有许多免费可得的版本可用。我们贯穿全书使用snprintf，也推荐你出于可靠性考虑在自己的程序中改用它来代替sprintf。

值得注意的是，许多网络入侵是由黑客通过发送数据，导致服务器对sprintf的调用使其缓冲区溢出而发生的。必须小心使用的函数还有gets、strcat和strcpy，通常应分别改为调用fgets、strncat和strncpy。更好的替代函数是后来才引入的strlcat和strlcpy，它们确保结果是正确终止的

字符串。编写安全的网络程序的更多技巧参见［Garfinkel, Schwartz, and Spafford 2003］的第23章。

终止连接

22 服务器通过调用close关闭与客户的连接。该调用引发正常的TCP连接终止序列：每个方向上发送一个FIN，每个FIN又由各自的对端确认。2.6节将详细讲述TCP的三路握手和用于终止一个TCP连接的4个TCP分组。

与上节查看客户程序一样，本节查看服务器程序也非常简略，具体细节留待本书以后论述。有以下几点需要注意。

与其客户程序一样，这一服务器程序也与IPv4协议相关。我们将在图11-13中给出使用getaddrinfo函数实现的一个协议无关的版本。

本服务器一次只能处理一个客户。如果多个客户连接差不多同时到达，系统内核在某个最大数目的限制下把它们排入队列，然后每次返回一个给accept函数。本服务器只需调用time和ctime这两个库函数，运行速度很快。然而如果服务器需用较多时间（譬如说几秒钟或一分钟）服务每个客户，那么我们必须以某种方式重叠对各个客户的服务。

图1-9中所示的服务器称为迭代服务器（iterative server），因为对于每个客户它都迭代执行一次。同时能处理多个客户的并发服务器（concurrent server）有多种编写技术。最简单的技术是调用Unix的fork函数（4.7节），为每个客户创建一个子进程。其他技术包括使用线程代替fork（26.4节），或在服务器启动时预先fork一定数量的子进程（30.6节）。

如果从shell命令行启动本例这样的一个服务器，我们也许想要它运行很长时间，因为服务器往往在系统工作期间一直运行。这要求我们往服务器程序中添加代码，以便它能够作为一个Unix守护进程（daemon）——能在后台运行且不跟任何终端关联的进程——运行。我们将在13.4节讨论守护进程。

1.6 本书中客户/服务器程序示例索引表

贯穿全书的用于阐述网络编程中使用的各种技术的两个客户/服务器程序示例如下：

时间获取客户/服务器程序（开始于图1-5、图1-6和图1-9）；

回射客户/服务器程序（开始于第5章）。

为了提供本书所涵盖不同主题的路线图，我们用下面4个表格汇总了将要开发的程序，并给出了它们的源代码所在的起始图号。图1-10列出了本书开发的时间获取客户程序的不同版本，其中有两个版本前面已讲过。图1-11列出了时间获取服务器程序的不同版本。图1-12列出了回射客户程序的不同版本，图1-13列出了回射服务器程序的不同版本。

图　　号	说　　明
1-5	TCP/IPv4，协议相关
1-6	TCP/IPv6，协议相关
11-4	TCP/IPv4，协议相关，调用gethostbyname和getservbyname
11-11	TCP，协议无关，调用getaddrinfo和tcp_connect
11-16	UDP，协议无关，调用getaddrinfo和udp_client
16-11	TCP，使用非阻塞connect
31-8	TCP，协议相关，用TPI取代套接字
E-1	TCP，协议相关，产生SIGPIPE
E-5	TCP，协议相关，输出套接字接收缓冲区的大小和MSS
E-11	TCP，协议相关，允许主机名（gethostbyname）或者IP地址
E-12	TCP，协议无关，允许主机名（gethostbyname）

图1-10 本书开发的时间获取客户程序的不同版本

图　　号	说　　明
1-9	TCP/IPv4，协议相关
11-13	TCP，协议无关，调用getaddrinfo和tcp_listen
11-14	TCP，协议无关，调用getaddrinfo和tcp_listen
11-19	UDP，协议无关，调用getaddrinfo和udp_server
13-5	TCP，协议无关，作为孤立的守护进程运行
13-12	TCP，协议无关，从inetd守护进程派生

图1-11 本书开发的时间获取服务器程序的不同版本

图　　号	说　　明
5-4	TCP/IPv4，协议相关
6-9	TCP，使用select
6-13	TCP，使用select并操纵缓冲区
8-7	UDP/IPv4，协议相关
8-9	UDP，验证服务器的地址
8-17	UDP，调用connect获取异步错误
14-2	UDP，使用SIGALRM信号在读服务器的应答时启动超时
14-4	UDP，使用select函数在读服务器的应答时启动超时
14-5	UDP，使用SO_RCVTIMEO套接字选项在读服务器的应答时启动超时
15-4	Unix域字节流，协议相关
15-6	Unix域数据报，协议相关

图1-12 本书开发的回射客户程序的不同版本

图　　号	说　　明
16-3	TCP，使用非阻塞I/O
16-10	TCP，使用两个进程（fork）
16-21	TCP，建立连接，然后发送RST
14-15	TCP，使用/dev/poll达成多路复用
14-18	TCP，使用kqueue达成多路复用
20-5	UDP，具有竞争状态的广播
20-6	UDP，具有竞争状态的广播
20-7	UDP，通过使用pselect消除了竞争状态的广播
20-9	UDP，通过使用sigsetjmp和siglongjmp消除了竞争状态的广播
20-10	UDP，通过在信号处理函数中使用IPC消除了竞争状态的广播
22-6	UDP，使用超时、重传和序列号实现可靠性
24-14	（第2版）UDP，使用带外数据对服务器心搏测试①
26-2	TCP，使用两个线程
27-6	TCP/IPv4，指定一条源路径
27-13	UDP/IPv6，指定一条源路径

图1-12 （续）

图　　号	说　　明
5-2	TCP/IPv4，协议相关
5-12	TCP/IPv4，协议相关，收拾终止了的子进程
6-21	TCP/IPv4，协议相关，使用select，单个进程处理所有客户
6-25	TCP/IPv4，协议相关，使用poll，单个进程处理所有客户
8-3	UDP/IPv4，协议相关
8-24	TCP和UDP/IPv4，协议相关，使用select
14-14	TCP，使用标准I/O函数库
15-3	Unix域字节流，协议相关
15-5	Unix域数据报，协议相关
15-15	Unix域字节流，带有从客户端传递凭证
22-4	UDP，接收目的地址和收取接口信息，截取数据报
22-15	UDP，捆绑所有接口地址
25-4	UDP，使用信号驱动的I/O
26-3	TCP，每个客户一个线程
26-4	TCP，每个客户一个线程，可移植的参数传递
27-6	TCP/IPv4，输出接收到的源路径
27-14	UDP/IPv6，输出并反转接收到的源路径
28-31	UDP，使用icmpd接收异步错误
E-15	UDP，捆绑所有接口地址

图1-13 本书开发的回射服务器程序的不同版本

1.7 OSI模型

描述一个网络中各个协议层的常用方法是使用国际标准化组织（International Organization for Standardization，ISO）的计算机通信开放系统互连（open systems interconnection，OSI）模型。这是一个七层模型，如图1-14所示。图中同时给出了它与网际协议族的近似映射。

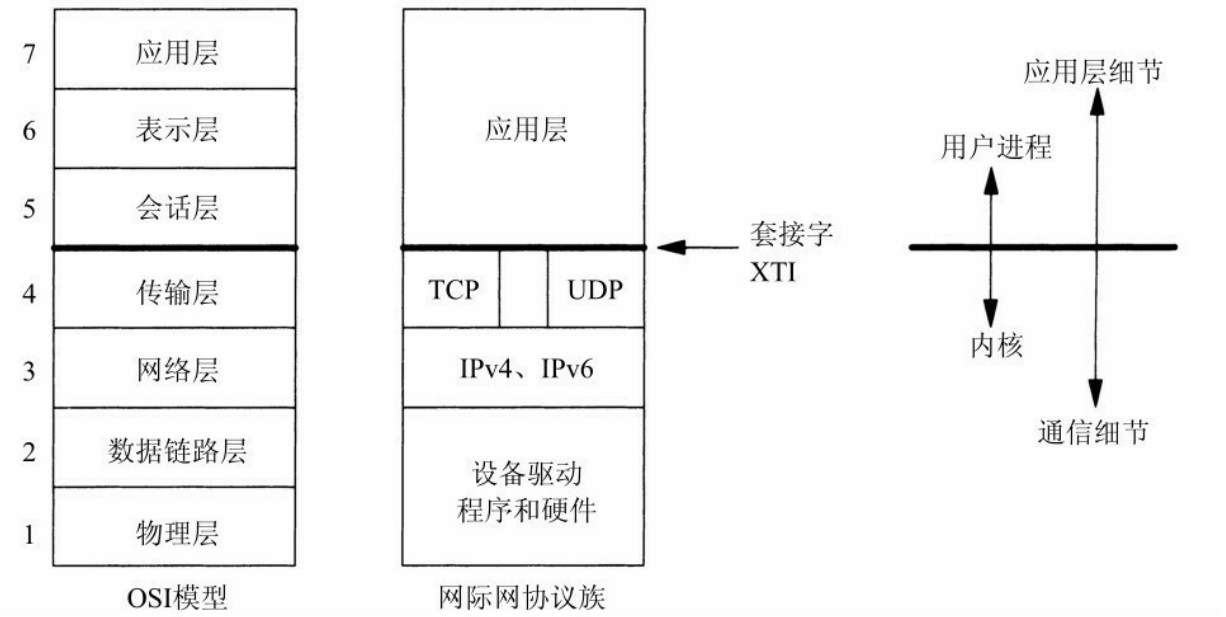


图1-14 OSI模型和网际协议族中的各层

我们认为OSI模型的底下两层是随系统提供的设备驱动程序和网络硬件。通常情况下，除需知道数据链路的某些特性外（如将在2.11节论述的1500字节以太网的MTU大小），我们不必关心这两层的具体情况。

网络层由IPv4和IPv6这两个协议处理，我们将在附录A中讲述它们。可以选择的传输层有TCP或UDP，我们将在第2章中讲述它们。图1-14中TCP与UDP之间留有间隙，表明网络应用绕过传输层直接使用IPv4或IPv6是可能的。这就是所谓的原始套接字（raw socket），我们将在第28章中讨论。

OSI模型的顶上三层被合并成一层，称为应用层。这就是Web客户（浏览器）、Telnet客户、Web服务器、FTP服务器和其他我们在使用的网络应用所在的层。对于网际协议，OSI模型的顶上三层协议几乎没有区别。

本书讲述的套接字编程接口是从顶上三层（网际协议的应用层）进入传输层的接口。本书的焦点是：如何使用套接字编写使用TCP或UDP

的网络应用程序。我们已提到原始套接字，在第29章中我们将看到，甚至可以彻底绕过IP层直接读写数据链路层的帧。

为什么套接字提供的是从OSI模型的顶上三层进入传输层的接口？这样设计有两个理由，如图1-14右侧所注。理由之一是顶上三层处理具体网络应用（如FTP、Telnet或HTTP）的所有细节，却对通信细节了解很少；底下四层对具体网络应用了解不多，却处理所有的通信细节：发送数据，等待确认，给无序到达的数据排序，计算并验证校验和，等等。理由之二是顶上三层通常构成所谓的用户进程（user process），底下四层却通常作为操作系统内核的一部分提供。Unix与其他现代操作系统都提供分隔用户进程与内核的机制。由此可见，第4层和第5层之间的接口是构建API的自然位置。 ■

1.8 BSD网络支持历史

套接字API起源于1983年发行的4.2BSD操作系统。图1-15展示了各种BSD发行版本的发展史，并注明了TCP/IP的主要发展历程。1990年面世的4.3BSD Reno发行版本随着OSI协议进入BSD内核而对套接字API做了少量的改动。

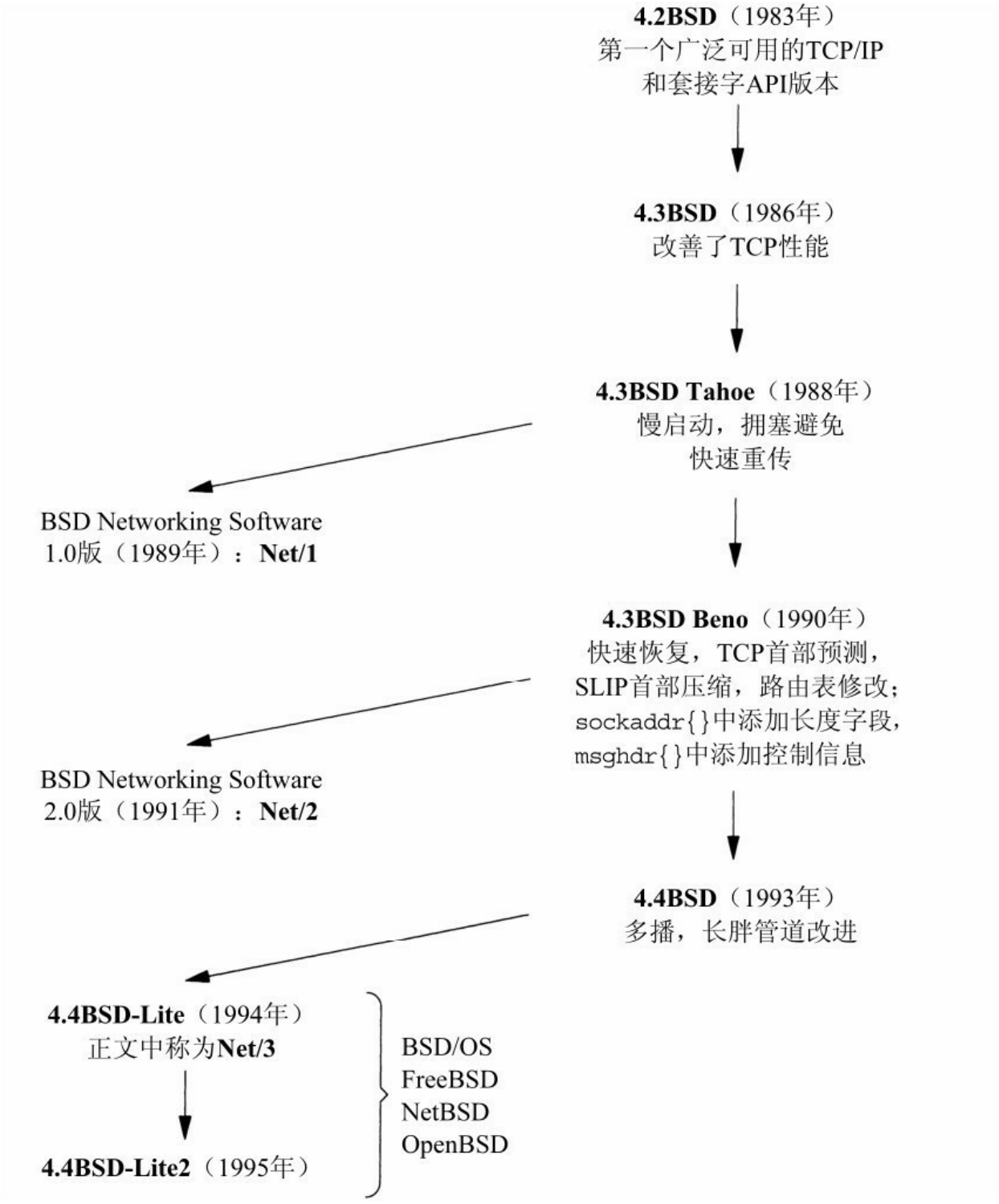


图1-15 各种BSD版本的历史

图1-15中从4.2BSD往下到4.4BSD的通路展示了源自Berkeley计算机系统研究组（Computer Systems Research Group，CSRG）的各个版本，

它们要求获取者已拥有Unix的源代码许可权。然而其中的所有网络支持代码，不论是内核支持（如TCP/IP协议栈、Unix域协议栈及套接字API）还是应用程序（如Telnet和FTP客户和服务器程序）都是独立于源自AT&T的Unix代码开发的。因此从1989年起，Berkeley开始提供第一个BSD网络支持版本，它包含所有的网络支持代码以及不受Unix源代码许可权约束的其他各种BSD系统软件。这些包含网络支持代码的版本是可公开获取的，最终因特网上任何人都可通过匿名FTP获取。

源自Berkeley的最终版本是1994年的4.4BSD-Lite和1995年的4.4BSD-Lite2。我们指出这两个版本是其他多个系统（包括BSD/OS、FreeBSD、NetBSD和OpenBSD）的基础，这些系统大多数仍然处于活跃的开发和完善之中。有关各种BSD版本和各种Unix系统历史的详情参见［Mckusick et al.1996］的第1章。 ■

许多Unix系统从某个版本的BSD网络支持代码（包括套接字API）开始提供网络支持，我们称这些实现为源自Berkeley的实现（Berkeley-derived implementation）。许多商业版本的Unix是基于System V版本4（System V Release 4，SVR4）的，其中有一些系统使用源自Berkeley的网络支持代码（如UnixWare 2.x），其他SVR4系统的网络支持代码却是独立起源的（如Solaris 2.x）。我们还要注意，Linux这种流行的可免费获得的Unix实现并不适合归属源自Berkeley的系列，因为它的网络支持代码和套接字API都是从头开始开发的。

1.9 测试用网络及主机 ■

图1-16展示了本书示例所用的各个网络和主机。对于每个主机，我们都标出了它的操作系统和硬件类型（因为有些操作系统可运行在不止一种硬件上）。各个框内的名字就是出现在本书中的各个主机名。

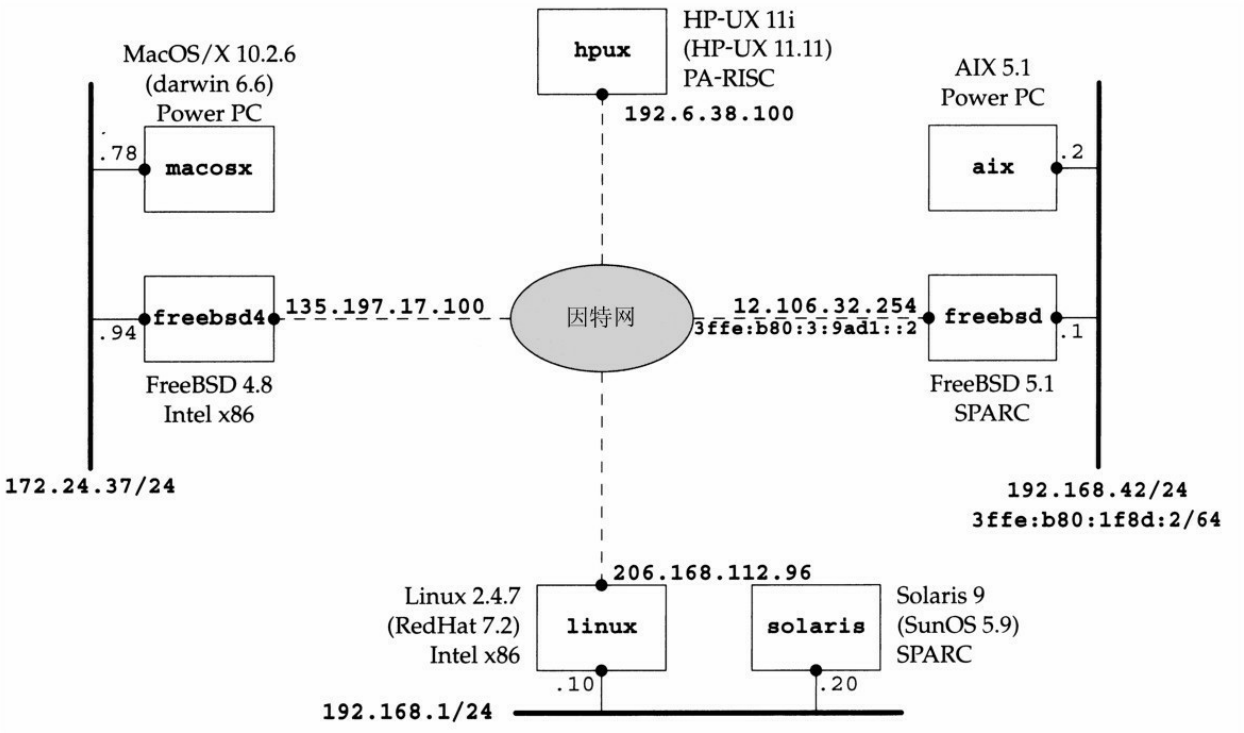


图1-16 本书示例所用的网络和主机

图1-16所示的拓扑适合本书的例子，不过机器大范围地散布在因特网上，物理拓扑实际上变得不太重要。事实上虚拟专用网络（virtual private network，VPN）或安全shell（secure shell， SSH）连接提供这些机器之间的连通性，而无需顾及这些主机的物理位置。

图中“/24”（和/64）指出从地址的最左位开始用于标识网络和子网的连续位数。A.4节将说明现今用于指定子网边界的/n记法。

Sun操作系统的真实名字是SunOS 5.x，而不是Solaris 2.x，但是大家习惯称它为Solaris，实际上这是操作系统和与之捆绑的其他软件的合称。

网络拓扑的发现

图1-16展示了本书的全部示例所用主机的网络拓扑，但是为了在你自己的网络上运行这些例子和完成习题，你可能需要了解自己的网络拓扑。尽管目前还没有关于网络配置和管理的现行Unix标准，但大多数Unix系统都提供了可用于发现某些网络细节的两个基本命令：netstat和

ifconfig。通过阅读所用系统上这些命令的手册页面 [9] ，你可以获悉有关它们的输出信息的详情。要留意的是，有些厂商把这些命令存放在诸如/sbin或/usr/sbin这样的管理目录中，而不是通常的/usr/bin目录，而这些管理目录可能不在通常的shell搜索路径中（由PATH环境变量指定）。

(1) netstat -i提供网络接口的信息。我们还指定-n标志以输出数值地址，而不是试图把它们反向解析成名字。下面的例子给出了接口及其名字和统计信息：

```
linux % netstat -ni
Kernel Interface table
Iface      MTU  Met     RX-OK RX-ERR RX-DRP RX-OVR     TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0      1500    0  49211085      0      0      0  40540958      0      0      0 BMRU
lo       16436    0  98613572      0      0      0  98613572      0      0      0 LRU
```

其中环回（loopback）接口称为lo，以太网接口称为eth0。下面的例子给出了支持IPv6的一个主机的类似信息：

```
freebsd % netstat -ni
Name       Mtu   Network       Address              Ipkts Ierrs     Opkts Oerrs  Coll
hme0      1500  <Link#1>      08:00:20:a7:68:6b  29100435    35  46561488     0     0
hme0      1500  12.106.32/24  12.106.32.254      28746630     -  46617260     -     -
hme0      1500 fe80:1::a00:20ff:fea7:686b/64
                                fe80:1::a00:20ff:fea7:686b
```

0 -
0 - -
hme0 1500 3ffe:b80:1f8d:1::1/64
3ffe:b80:1f8d:1::1 0 -
0 - -
hme1 1500 <Link#2> 08:00:20:a7:68:6b 51092
0 31537 0 0
hme1 1500 fe80:2::a00:20ff:fea7:686b/64
fe80:2::a00:20ff:fea7:686b
0 -
90 - -
hme1 1500 192.168.42 192.168.42.1
43584 - 24173 - -
hme1 1500 3ffe:b80:1f8d:2::1/64
3ffe:b80:1f8d:2::1 78 -
8 - -
lo0 16384 <Link#6>
10198 0 10198 0 0
lo0 16384 ::1/128 ::1
10 - 10 - -
lo0 16384 fe80:6::1/64 fe80:6::1
0 - 0 - -
lo0 16384 127 127.0.0.1
10167 - 10167 - -
gif0 1280 <Link#8>
6 0 5 0 0
gif0 1280 3ffe:b80:3:9ad1::2/128

3ffe:b80:3:9ad1::2

0 - 0 - -

gif0 1280 fe80:8::a00:20ff:fea7:686b/64

fe80:8::a00:20ff:fea7:686b

0 -

0 - -

注意：为了对齐输出字段，我们对较长的代码行做了回行处理。

(2) netstat –r展示路由表，也是另一种确定接口的方法。我们通常指定-n标志以输出数值地址。它还给出默认路由器的IP地址。

freebsd % netstat -nr

Routing tables

Internet:

Destination Gateway Flags Refs Use Netif Expire

default 12.106.32.1 USGc 10 6877 hme0

12.106.32/24 link#1 UC 3 0 hme0

12.106.32.1 00:b0:8e:92:2c:00 UHLW 9 7 hme0 1187

12.106.32.253 08:00:20:b8:f7:e0 UHLW 0 1 hme0 140

12.106.32.254 08:00:20:a7:68:6b UHLW 0 2 lo0

127.0.0.1 127.0.0.1 UH 1 10167 lo0

192.168.42 link#2 UC 2

0 hme1

192.168.42.1 08:00:20:a7:68:6b UHLW 0 11 lo0

192.168.42.2 00:04:ac:17:bf:38 UHLW 2 24108 hme1 210

Internet6:

Destination Gateway Flags Netif Expire

::/96 ::1 UGRSc lo0 =>

default 3ffe:b80:3:9ad1::1 UGSc gif0

::1 ::1 UH lo0

::ffff:0.0.0.0/96 ::1 UGRSc lo0

3ffe:b80:3:9ad1::1 3ffe:b80:3:9ad1::2 UH gif0

3ffe:b80:3:9ad1::2 link#8 UHL lo0

3ffe:b80:1f8d::/48 lo0 USc lo0

3ffe:b80:1f8d:1::/64 link#1 UC hme0

3ffe:b80:1f8d:1::1 08:00:20:a7:68:6b UHL lo0

3ffe:b80:1f8d:2::/64 link#2

UC hme1

3ffe:b80:1f8d:2::1 08:00:20:a7:68:6b

UHL lo0

3ffe:b80:1f8d:2:204:acff:fe17:bf38 00:04:ac:17:bf:38

UHLW hme1

fe80::/10 ::1

UGRSc lo0

fe80::%hme0/64

link#1 UC hme0

fe80::a00:20ff:fea7:686b%hme0 08:00:20:a7:68:6b

UHL lo0

fe80::%hme1/64

link#2 UC hme1

fe80::a00:20ff:fea7:686b%hme1 08:00:20:a7:68:6b

UHL lo0

fe80::%lo0/64 fe80::1%lo0

Uc lo0

fe80::1%lo0

link#6 UHL lo0

fe80::%gif0/64

link#8 UC gif0

fe80::a00:20ff:fea7:686b%gif0 link#8

UHL lo0

ff01::/32 ::1

U lo0

ff02::/16 ::1

UGRS lo0

ff02::%hme0/32

link#1 UC hme0

ff02::%hme1/32

link#2 UC hme1

ff02::%lo0/32

::1 UC lo0

ff02::%gif0/32

link#8 UC gif0

(3) 有了各个网络接口的名字，执行ifconfig就可获得每个接口的详细信息。

linux % ifconfig eth0

eth0 Link encap:Ethernet HWaddr 00:C0:9F:06:B0:E1

inet addr:206.168.112.96 Bcast:206.168.112.127 Mask:255.255.255.128

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:49214397 errors:0 dropped:0 overruns:0 frame:0

TX packets:40543799 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:100

RX bytes:1098069974 (1047.2 Mb) TX bytes:3360546472 (3204.8 Mb)

Interrupt:11 Base address:0x6000

该命令给出了指定接口的IP地址、子网掩码和广播地址。其中的MULTICAST标志通常指明该接口所在主机支持多播。有些ifconfig的实现还提供-a标志，用于输出所有已配置接口的信息。

(4)　找出本地网络中众多主机的IP地址的方法之一是，针对从上一步找到的本地接口的广播地址执行ping命令。

linux % ping -b 206.168.112.127

WARNING: pinging broadcast address

PING 206.168.112.127 (206.168.112.127) from 206.168.112.96 : 56(84) bytes of data.

64 bytes from 206.168.112.96: icmp_seq=0 ttl=255 time=241 usec

64 bytes from 206.168.112.40: icmp_seq=0 ttl=255 time=2.566 msec (DUP!)

64 bytes from 206.168.112.118: icmp_seq=0 ttl=255 time=2.973 msec (DUP!)

64 bytes from 206.168.112.14: icmp_seq=0 ttl=255 time=3.089 msec (DUP!)

64 bytes from 206.168.112.126: icmp_seq=0 ttl=255 time=3.200 msec (DUP!)

64 bytes from 206.168.112.71: icmp_seq=0 ttl=255 time=3.311 msec (DUP!)

64 bytes from 206.168.112.31: icmp_seq=0 ttl=64 time=3.541 msec (DUP!)

64 bytes from 206.168.112.7: icmp_seq=0 ttl=255 time=3.636 msec (DUP!)

...

1.10 Unix标准

在编写本书时，最引人注目的Unix标准化活动是由Austin公共标准修订组（The Austin Common Standards Revision Group，CSRG）主持

的。他们的努力结果是涵盖1 700多个编程接口的约4 000页内容的规范［Josey 2002］。这些规范既具有IEEE POSIX名字，也具有开放团体的技术标准（The Open Group’s Technical Standard）名字。其结果是同一个Unix标准有多个名字来指称：ISO/IEC 9945:2002、IEEE Std 1003.1-2001和单一Unix规范第3版（Single Unix Specification Version 3）都指同一个标准。本书中除了像本节这样需要讨论各种较早期标准各自特性的章节外，我们简单地称这个Unix标准为POSIX规范（The POSIX Specification）。

获取这个统一标准的最简易方法是定购其CD-ROM拷贝或通过Web免费访问。这两种方法的起始点都是http://www.UNIX.org/version3。

1.10.1 POSIX的背景

POSIX（可移植操作系统接口）是Portable Operating System Interface的首字母缩写。它并不是单个标准，而是由电气与电子工程师学会（the Institute for Electrical and Electronics Engineers, Inc.）即IEEE开发的一系列标准。POSIX标准已被国际标准化组织即ISO和国际电工委员会（the International Electrotechnical Commission）即IEC采纳为国际标准（这两个组织合称为ISO/IEC）。下面是POSIX标准的发展简史。

第一个POSIX标准是IEEE Std 1003.1-1988（317页）。它详述了进入类Unix内核的C语言接口，涵盖了下述领域：进程原语（fork、exec、信号和定时器）、进程环境（用户ID和进程组）、文件与目录（所有I/O函数）、终端I/O、系统数据库（口令文件和用户组文件）以及tar和cpio归档格式。

第一个POSIX标准在1986年是称为“IEEE-IX”的试用版。POSIX这个名字是由Richard Stallman建议使用的。

第二个POSIX标准是IEEE Std 1003.1-1990（356页），也称为ISO/IEC 9945-1: 1990。从1988版本到1990版本只做了少量的修改。新添的副标题为“Part 1: System Application Program Interface (API) [C

Language]”，表明本标准为C语言API。

下一个标准是两卷本的IEEE Std 1003.2-1992（约1300页）。它的副标题为“Part 2: Shell and Utilities”。这一部分定义了shell（基于System V的Bourne Shell）和大约100个实用程序（通常从shell启动执行的程序，如awk、basename、vi和yacc等等）。本书称这个标准为POSIX.2。

再下一个标准是IEEE Std 1003.1b-1993（590页），先前称为IEEE P1003.4。这是对1003.1-1990标准的更新，添加了由P1003.4工作组开发的实时扩展。1003.1b-1993相比1990年版标准新增的条目包括：文件同步、异步I/O、信号量、存储管理（mmap和共享内存）、执行调度、时钟与定时器以及消息队列。

更下一个标准是IEEE Std 1003.1 1996年版［IEEE 1996］（743页），也称为ISO/IEC 9945-1:1996，它包括1003.1-1990（基本API）、1003.1b-1993（实时扩展）、1003.1c-1995 （pthreads）和1003.1i-1995（对1003.1b的技术性修订）。该标准增添了3章关于线程的内容，并另有关于线程同步（互斥锁和条件变量）、线程调度和同步调度的各节。本书称这个标准为POSIX.1。该标准还有一个前言，其中声明ISO/IEC 9945由下面3个部分构成。

Part 1: System API (C language)——第1部分：系统API（C语言）。

Part 2: Shell and utilities——第2部分：Shell和实用程序。

Part 3: System administration——第3部分：系统管理（正在开发中）。

第1部分和第2部分就是我们所说的POSIX.1和POSIX.2。

743页中有超过四分之一的篇幅是一个标题为“Rationale and Notes”（理由与注解）的附录。该附录含有历史性信息和某些特性被加入或删除的理由。这些理由通常跟正式标准一样有教益。

最后一个标准是在2000年被认可 [10] 的IEEE Std 1003.1g: Protocol-independent interfaces (PII)。在单一Unix规范第3版（The Single Unix

Specification Version 3）面世之前，这是与本书涵盖的主题最为相关的POSIX产品。它是联网API标准，它定义了两个API，并称它们为详尽网络接口（Detailed Network Interface，DNI）。

DNI/Socket，基于4.4BSD的套接字API。

DNI/XTI，基于X/Open的XPG4规范。

这个标准的工作作为P1003.12工作组（后来改名为P1003.1g）起始于20世纪80年代后期。本书称这个标准为POSIX.1g。

关于各种POSIX标准的当前状况可以访问http://www.pasc.org/standing/sd11.html。

1.10.2 开放团体的背景

开放团体（The Open Group）是由1984年成立的X/Open公司（X/Open Company）和1988年成立的开放软件基金会（Open Software Foundation，OSF）于1996年合并成的组织。它是厂商、工业界最终用户、政府和学术机构共同参加的国际组织。下面是开放团体制定的标准的简要背景。

X/Open公司于1989年出版了X/Open Portability Guide（X/Open移植性指南，XPG）第3期，即XPG3。

XPG第4期即XPG4出版于1992年，其第2版出版于1994年。这个最新版本也称为“Spec 1 170”，其中魔数1170是系统接口数（926个）、头文件数（70个）和命令数（174个）的总和。这组规范的最终名字是X/Open Single Unix Specification（X/Open单一Unix规范），也称为“Unix 95”。

单一Unix规范第2版于1997年3月发行。符合这个规范的产品称为“Unix 98”。本书就称这个规范为“Unix 98”。Unix 98的接口数目从1170个增长到1434个，而用于工作站的接口数则达到3 030个，因为它包含公共桌面环境（Common Desktop Environment，CDE），而公共桌面环境又需要X Windows系统和Motif用户接口。本规范的详情参见

http://www.UNIX.org/version2和［Josey 1997］。Unix 98为套接字API和XTI API定义了网络支持服务。这个规范与POSIX.1g几乎相同。

不幸的是，X/Open称它们的网络标准为XNS：X/Open Networking Services。定义Unix 98套接字和XTI的文档的这一版本称为“XNS Issue 5”（XNS第5期）。在网络界，XNS已是Xerox Network Systems体系结构的简称。所以，我们避免使用XNS，而称这个X/Open文档为Unix 98网络API标准。

1.10.3 标准的统一

如本节开头所提，伴随Austin CSRG发布单一Unix规范第3版，POSIX和开放团体都继续发展，达成统一的标准。CSRG促成50多家公司就单一标准达成一致意见，这在Unix发展史上确实是一件划时代之大事。如今大多数Unix系统都符合POSIX.1和POSIX.2的某个版本，不少系统符合单一Unix规范第3版。

历史上多数Unix系统或者源自Berkeley，或者源自System V，不过这些差别在慢慢消失，因为大多数厂商已开始采纳这些标准。然而在系统管理的处理上两者仍然存在较大差别，这个领域目前还没有标准可循。

本书的焦点是单一Unix规范第3版，其中又以套接字API为主。只要可能，我们就使用标准函数。

1.10.4 因特网工程任务攻坚组

因特网工程任务攻坚组（Internet Engineering Task Force，IETF）是一个由关心因特网体系结构的发展及其顺利运作的网络设计者、操作员、厂商和研究人员联合组成的开放的国际团体。它向任何感兴趣的个人开放。

因特网标准处理过程在RFC 2026［Bradner 1996］中说明。因特网标准一般处理协议问题而不是编程API，不过仍有两个RFC（RFC 3493［Gilligan et al. 2003］和RFC 3542［Stevens et al. 2003］）说明了

IPv6的套接字API。它们是信息性的RFC，并不是标准，制定它们的目的是加速部署由多家从事IPv6工作较早的厂商所开发的可移植网络应用程序。尽管标准主体趋于花费很长的时间，其中许多API却已经在单一Unix规范第3版中标准化了。

1.11 64位体系结构

20世纪90年代中期到末期开始出现向64位体系结构和64位软件发展的趋势。其原因之一是在每个进程内部可以由此使用更长的编址长度（即64位指针），从而可以寻址很大的内存空间（超过2^{32} 字节）。现有32位Unix系统上共同的编程模型称为ILP32模型，表示整数（I）、长整数（L）和指针（P）都占用32位。64位Unix系统上变得最为流行的模型称为LP64模型，表示只有长整数（L）和指针（P）占用64位。图1-17对这两种模型进行了比较。

从编程角度看，LP64模型意味着我们不能假设一个指针能存放在一个整数中。我们还必须考虑LP64模型对现有API的影响。

数据类型	ILP32模型	LP64模型
char	8	8
short	16	16
int	32	32
long	32	64
指针	32	64

图1-17 ILP32和LP64模型保存不同数据类型所占用的位数的比较

ANSI C创造了size_t数据类型，它用于作为malloc的唯一参数（待分配的字节数），或者作为read和write的第三个参数（待读或写的字节数）。在32位系统中size_t是一个32位值，但是在64位系统中它必须是

一个64位值，以便发挥更大寻址模型的优势。这意味着64位系统中也许含有一个把size_t定义为unsigned long的typedef指令。联网API存在如下问题：POSIX.1g的某些草案规定，存放套接字地址结构大小的函数参数具有size_t数据类型（如bind和connect的第三个参数）。某些XTI结构也含有数据类型为long的成员（如t_info和t_opthdr结构）。如果这些规定不加修改，当Unix系统从ILP32模型转变为LP64模型时，size_t和long都将从32位值变为64位值。这两个例子实际上并不需要使用64位的数据类型：套接字地址结构的长度最多也就几百个字节，给XTI的结构成员使用long数据类型则是个错误。

处理这些情况的办法是使用专门设计的数据类型。套接字API对套接字地址结构的长度使用socklen_t数据类型，XTI则使用t_scalar_t和t_uscalar_t数据类型。不把这些值由32位改为64位的理由是易于为那些已在32位系统中编译的应用程序提供在新的64位系统中的二进制代码兼容性。

1.12 小结 ■

图1-5展示了一个尽管简单但却完整的TCP客户程序，它从某个指定的服务器读取当前时间和日期；而图1-9则展示了其服务器程序的一个完整版本。这两个例子引入了许多本书其他部分将要扩展的概念和术语。

我们的客户程序与IPv4协议相关，我们于是把它修改成使用IPv6，但这样做却只是给了我们另外一个协议相关的程序。我们将在第11章中开发一些可用来编写协议无关代码的函数，这在因特网开始使用IPv6后会变得非常重要。

纵贯本书，我们将使用1.4节中介绍的包裹函数来缩短代码，同时又保证测试每个函数调用，检查是否返回错误。我们的包裹函数都以一

个大写字母开头。

单一Unix规范第3版有多个名称，我们简单地称之为POSIX规范。它是两个长期发展的标准团体各自努力的汇合，由Austin　CSRG最终团结起来。

对Unix网络支持历史感兴趣的读者可参阅叙述Unix历史的［Salus 1994］和叙述TCP/IP及因特网历史的［Salus 1995］。

习题

1.1 按1.9节末尾的步骤找出你自己的网络拓扑的信息。

1.2　获取本书示例的源代码（见前言），编译并测试图1-5所示的TCP时间获取客户程序。运行这个程序若干次，每次以不同IP地址作为命令行参数。

1.3　把图1-5中的socket的第一参数改为9999。编译并运行这个程序。结果如何？找出对应于所输出出错的errno值。你如何可以找到关于这个错误的更多信息？

1.4 修改图1-5中的while循环，加入一个计数器，累计read返回大于零值的次数。在终止前输出这个计数器值。编译并运行你的新客户程序。

1.5 按下述步骤修改图1-9中的程序。首先，把赋于sin_port的端口号从13改为9999。然后，把write的单一调用改为循环调用，每次写出结果字符串的一个字节。编译修改后的服务器程序并在后台启动执行。接着修改前一道习题中的客户程序（它在终止前输出计数器值），把赋于sin_port的端口号从13改为9999。启动这个客户程序，指定运行修改后的服务器程序的主机的IP地址作为命令行参数。客户程序计数器的输出值是多少？如果可能，在不同主机上运行这个客户与服务器程序。

第2章 传输层：TCP、UDP和SCTP

2.1 概述

本章提供本书示例所用TCP/IP协议的概貌。我们的目的是从网络编程角度提供足够的细节以理解如何使用这些协议，同时提供有关这些协议的实际设计、实现及历史的具体描述的参考点。

本章的焦点是传输层，包括TCP、UDP和SCTP（Stream Control Transmission Protocol，流控制传输协议）。绝大多数客户/服务器网络应用使用TCP或UDP。SCTP是一个较新的协议，最初设计用于跨因特网传输电话信令。这些传输协议都转而使用网络层协议IP：或是IPv4，或是IPv6。尽管可以绕过传输层直接使用IPv4或IPv6，但这种技术（往往称为原始套接字）却极少使用。因此，我们把IPv4和IPv6以及ICMPv4和ICMPv6的详细描述安排在附录A中。

UDP是一个简单的、不可靠的数据报协议，而TCP是一个复杂、可靠的字节流协议。SCTP与TCP类似之处在于它也是一个可靠的传输协议，但它还提供消息边界、传输级别多宿（multihoming）支持以及将头端阻塞（head-of-line blocking）减少到最小的一种方法。我们必须了解由这些传输层协议提供给应用进程的服务，这样才能弄清这些协议处理什么，应用进程中又需要处理什么。

TCP的某些特性一旦理解，就很容易编写健壮的客户和服务器程序，也很容易使用诸如netstat等普遍可用的工具来调试客户和服务器程序。本章将阐述以下相关主题：TCP的三路握手、TCP的连接终止序列和TCP的TIME_WAIT状态，SCTP的四路握手和SCTP的连接终止，加上由套接字层提供的TCP、UDP和SCTP缓冲机制，等等。

2.2 总图

虽然协议族被称为“TCP/IP”，但除了TCP和IP这两个主要协议外，还有许多其他成员。图2-1展示了这些协议的概况。

图2-1中同时展示了IPv4和IPv6。从右向左查看该图，最右边的5个网络应用在使用IPv6；我们将在第3章中随sockaddr_in6结构讲解AF_INET6常值。随后的6个网络应用使用IPv4。

最左边名为tcpdump的网络应用或者使用BSD分组过滤器（BSD packet filter，BPF），或者使用数据链路提供者接口（datalink provider interface，DLPI）直接与数据链路进行通信。处于其右边所有9个应用下面的虚线标记为API，它通常是套接字或XTI。访问BPF或DLPI的接口不使用套接字或XTI。

这种情况存在一个例外：Linux使用一种称为SOCK_PACKET的特殊套接字类型提供对于数据链路的访问。我们将在第28章中详细讲述这个例外。

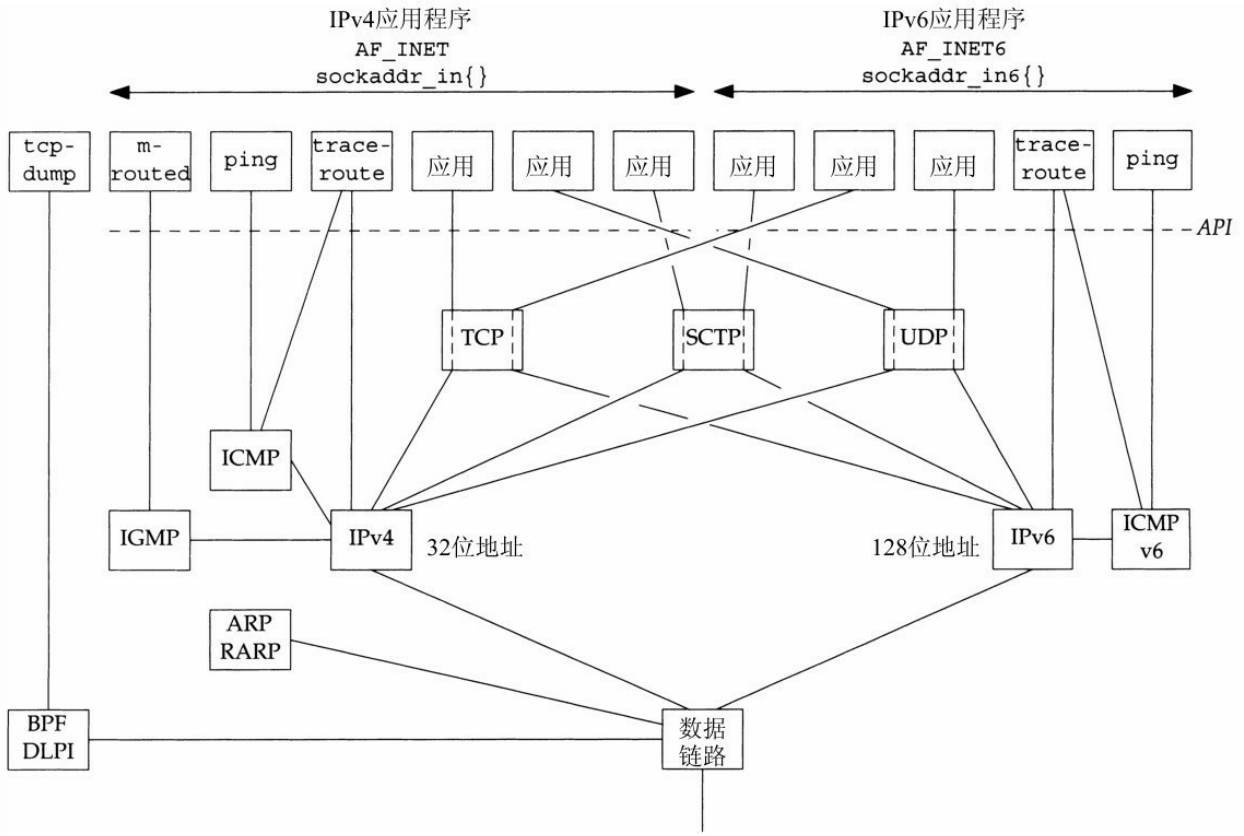


图2-1 TCP/IP协议概况

图2-1中还标明traceroute程序使用两种套接字：IP套接字用于访问IP，ICMP套接字用于访问ICMP。在第28章中，我们将开发ping和traceroute这两个应用的IPv4和IPv6版本。

下面我们讲解一下图2-1中的每一个协议框。

IPv4 网际协议版本4（Internet Protocol version 4）。IPv4（通常称之为IP）自20世纪80年代早期以来一直是网际协议族的主力协议。它使用32位地址（见A.4节）。IPv4给TCP、UDP、SCTP、ICMP和IGMP提供分组递送服务。

IPv6 网际协议版本6（Internet Protocol version 6）。IPv6是在20世纪90年代中期作为IPv4的一个替代品设计的。其主要变化是使用128位更大地址（见A.5节）以应对20世纪90年代因特网的爆发性增长。IPv6给TCP、UDP、SCTP和ICMPv6提供分组递送服务。

当无需区别IPv4和IPv6时，我们经常把“IP”一词作为形容词使用，如IP层、IP地址等。

TCP 传输控制协议（Transmission Control Protocol）。TCP是一个面向连接的协议，为用户进程提供可靠的全双工字节流。TCP套接字是一种流套接字（stream socket）。TCP关心确认、超时和重传之类的细节。大多数因特网应用程序使用TCP。注意，TCP既可以使用IPv4，也可以使用IPv6。

UDP 用户数据报协议（User Datagram Protocol）。UDP是一个无连接协议。UDP套接字是一种数据报套接字（datagram socket）。UDP数据报不能保证最终到达它们的目的地。与TCP一样，UDP既可以使用IPv4，也可以使用IPv6。

SCTP 流控制传输协议（Stream Control Transmission Protocol）。SCTP是一个提供可靠全双工关联的面向连接的协议，我们使用“关联”一词来指称SCTP中的连接，因为SCTP是多宿的，从而每个关联的两端均涉及一组IP地址和一个端口号。SCTP提供消息服务，也就是维护来自应用层的记录边界。与TCP和UDP一样，SCTP既可以使用IPv4，也可以使用IPv6，而且能够在同一个关联中同时使用它们。

ICMP 网际控制消息协议（Internet Control Message Protocol）。ICMP处理在路由器和主机之间流通的错误和控制消息。这些消息通常由TCP/IP网络支持软件本身（而不是用户进程）产生和处理，不过图中展示的ping和traceroute程序同样使用ICMP。有时我们称这个协议为ICMPv4，以便与ICMPv6相区别。

IGMP 网际组管理协议（Internet Group Management Protocol）。IGMP用于多播（见第21章），它在IPv4中是可选的。

ARP 地址解析协议（Address Resolution Protocol）。ARP把一个IPv4地址映射成一个硬件地址（如以太网地址）。ARP通常用于诸如以太网、令牌环网和FDDI等广播网络，在点到点网络上并不需要。

RARP 反向地址解析协议（Reverse Address Resolution Protocol）。RARP把一个硬件地址映射成一个IPv4地址。它有时用于无盘节点的引导。

ICMPv6 网际控制消息协议版本6（Internet Control Message Protocol version 6）。ICMPv6综合了ICMPv4、IGMP和ARP的功能。

BPF BSD分组过滤器（BSD packet filter）。该接口提供对于数据链路层的访问能力，通常可以在源自Berkeley的内核中找到。

DLPI 数据链路提供者接口（datalink provider interface）。该接口也提供对于数据链路层的访问能力，通常随SVR4内核提供。

所有网际协议由一个或多个称为请求评注（Request for Comments，RFC）的文档定义，这些RFC就是它们的正式规范。习题2.1的答案说明如何获得这些RFC。

我们使用术语“IPv4/IPv6主机”或“双栈主机”表示同时支持IPv4和IPv6的主机。

TCP/IP协议的其他细节参见TCPv1。TCP/IP在4.4BSD上的实现参见TCPv2。

2.3 用户数据报协议（UDP）

UDP是一个简单的传输层协议，在RFC 768［Postel 1980］中有详细说明。应用进程往一个UDP套接字写入一个消息，该消息随后被封装（encapsulating）到一个UDP数据报，该UDP数据报进而又被封装到一个IP数据报，然后发送到目的地。UDP不保证UDP数据报会到达其最终目的地，不保证各个数据报的先后顺序跨网络后保持不变，也不保证每个数据报只到达一次。

我们使用UDP进行网络编程所遇到的问题是它缺乏可靠性。如果一个数据报到达了其最终目的地，但是校验和检测发现有错误，或者该数

据报在网络传输途中被丢弃了，它就无法被投递给UDP套接字，也不会被源端自动重传。如果想要确保一个数据报到达其目的地，可以往应用程序中添置一大堆的特性：来自对端的确认、本端的超时与重传等。

每个UDP数据报都有一个长度。如果一个数据报正确地到达其目的地，那么该数据报的长度将随数据一道传递给接收端应用进程。我们已经提到过TCP是一个字节流（byte-stream）协议，没有任何记录边界（见1.2节），这一点不同于UDP。

我们也说UDP提供无连接的（connectionless）服务，因为UDP客户与服务器之间不必存在任何长期的关系。举例来说，一个UDP客户可以创建一个套接字并发送一个数据报给一个给定的服务器，然后立即用同一个套接字发送另一个数据报给另一个服务器。同样地，一个UDP服务器可以用同一个UDP套接字从若干个不同的客户接收数据报，每个客户一个数据报。

2.4 传输控制协议（TCP）

由TCP向应用进程提供的服务不同于由UDP提供的服务。TCP在RFC 793［Poste1 1981c］中有详细说明，然后由RFC 1323［Jacobson, Braden, and Borman 1992］、RFC 2581［Allman, Paxson, and Stevens 1999］、RFC 2988［Paxson and Allman 2000］和RFC 3390［Allman, Floyd, and Partridge 2002］加以更新。首先，TCP提供客户与服务器之间的连接（connection）。TCP客户先与某个给定服务器建立一个连接，再跨该连接与那个服务器交换数据，然后终止这个连接。

其次，TCP还提供了可靠性（reliability）。当TCP向另一端发送数据时，它要求对端返回一个确认。如果没有收到确认，TCP就自动重传数据并等待更长时间。在数次重传失败后，TCP才放弃，如此在尝试发送数据上所花的总时间一般为4～10分钟（依赖于具体实现）。

注意，TCP并不保证数据一定会被对方端点接收，因为这是不可能做到的。如果有可能， TCP就把数据递送到对方端点，否则就（通过放弃重传并中断连接这一手段）通知用户。这么说来，TCP也不能被描述成是100%可靠的协议，它提供的是数据的可靠递送或故障的可靠通知。

TCP含有用于动态估算客户和服务器之间的往返时间（round-trip time，RTT）的算法，以便它知道等待一个确认需要多少时间。举例来说，RTT在一个局域网上大约是几毫秒，跨越一个广域网则可能是数秒钟。另外，因为RTT受网络流通各种变化因素影响，TCP还持续估算一个给定连接的RTT。

TCP通过给其中每个字节关联一个序列号对所发送的数据进行排序（sequencing）。举例来说，假设一个应用写2048字节到一个TCP套接字，导致TCP发送2个分节：第一个分节所含数据的序列号为1～1024，第二个分节所含数据的序列号为1025～2048。（分节是TCP传递给IP的数据单元。）如果这些分节非顺序到达，接收端TCP将先根据它们的序列号重新排序，再把结果数据传递给接收应用。如果接收端TCP接收到来自对端的重复数据（譬如说对端认为一个分节已丢失并因此重传，而这个分节并没有真正丢失，只是网络通信过于拥挤），它可以（根据序列号）判定数据是重复的，从而丢弃重复数据。

UDP不提供可靠性。UDP本身不提供确认、序列号、RTT估算、超时和重传等机制。如果一个UDP数据报在网络中被复制，两份副本就可能都递送到接收端的主机。同样地，如果一个UDP客户发送两个数据报到同一个目的地，它们可能被网络重新排序，颠倒顺序后到达目的地。UDP应用必须处理所有这些情况，在22.5节中我们将展示如何处理。

再次，TCP提供流量控制（flow　control）。TCP总是告知对端在任何时刻它一次能够从对端接收多少字节的数据，这称为通告窗口（advertised　window）。在任何时刻，该窗口指出接收缓冲区中当前可

用的空间量，从而确保发送端发送的数据不会使接收缓冲区溢出。该窗口时刻动态变化：当接收到来自发送端的数据时，窗口大小就减小，但是当接收端应用从缓冲区中读取数据时，窗口大小就增大。通告窗口大小减小到0是有可能的：当TCP对应某个套接字的接收缓冲区已满，导致它必须等待应用从该缓冲区读取数据时，方能从对端再接收数据。

UDP不提供流量控制。如我们将在8.13节所示，让较快的UDP发送端以一个UDP接收端难以跟上的速率发送数据报是非常容易的。

最后，TCP连接是全双工的（full-duplex）。这意味着在一个给定的连接上应用可以在任何时刻在进出两个方向上既发送数据又接收数据。因此，TCP必须为每个数据流方向跟踪诸如序列号和通告窗口大小等状态信息。建立一个全双工连接后，需要的话可以把它转换成一个单工连接（见6.6节）。

UDP可以是全双工的。

2.5 流控制传输协议（SCTP）

SCTP提供的服务与UDP和TCP提供的类似。SCTP在RFC 2960［Stewart et al. 2000］中详细说明，并由RFC 3309［Stone, Stewart, and Otis 2002］加以更新。RFC 3286［Ong and Yoakum 2002］给出了SCTP的简要介绍。SCTP在客户和服务器之间提供关联（association），并像TCP那样给应用提供可靠性、排序、流量控制以及全双工的数据传送。SCTP中使用“关联”一词取代“连接”是为了避免这样的内涵：一个连接只涉及两个IP地址之间的通信。一个关联指代两个系统之间的一次通信，它可能因为SCTP支持多宿而涉及不止两个地址。

与TCP不同的是，SCTP是面向消息的（message-oriented）。它提供各个记录的按序递送服务。与UDP一样，由发送端写入的每条记录的

长度随数据一道传递给接收端应用。

SCTP能够在所连接的端点之间提供多个流，每个流各自可靠地按序递送消息。一个流上某个消息的丢失不会阻塞同一关联其他流上消息的投递。这种做法与TCP正好相反，就TCP而言，在单一字节流中任何位置的字节丢失都将阻塞该连接上其后所有数据的递送，直到该丢失被修复为止。

SCTP还提供多宿特性，使得单个SCTP端点能够支持多个IP地址。该特性可以增强应对网络故障的健壮性。一个端点可能有多个冗余的网络连接，每个网络又可能有各自接入因特网基础设施的连接。当该端点与另一个端点建立一个关联后，如果它的某个网络或某个跨越因特网的通路发生故障，SCTP就可以通过切换到使用已与该关联相关的另一个地址来规避所发生的故障。

类似的健壮性在路由协议的辅助下也可以从TCP中获得。举例来说，由iBGP实现的同一域内的BGP连接往往把赋予路由器内某个虚拟接口的多个地址用作TCP连接的端点。该域的路由协议确保两个路由器之间只要存在一条路由，该路由就会被用上，从而保证这两个路由器之间的BGP连接可用；要是使用属于某个物理接口的地址来建立BGP连接，该物理接口又变得不工作了，这一点就不可能做到。SCTP的多宿特性允许主机（而不仅仅是路由器）也多宿，而且允许多宿跨越不同的服务供应商发生，这些基于路由的TCP多宿方法都无法做到。

2.6 TCP连接的建立和终止

为帮助大家理解connect、accept和close这3个函数并使用netstat程序调试TCP应用，我们必须了解TCP连接如何建立和终止，并掌握TCP的状态转换图。

2.6.1 三路握手

建立一个TCP连接时会发生下述情形。

(1)　服务器必须准备好接受外来的连接。这通常通过调用socket、bind和listen这3个函数来完成，我们称之为被动打开（passive open）。

(2) 客户通过调用connect发起主动打开（active open）。这导致客户TCP发送一个SYN（同步）分节，它告诉服务器客户将在（待建立的）连接中发送的数据的初始序列号。通常SYN分节不携带数据，其所在IP数据报只含有一个IP首部、一个TCP首部及可能有的TCP选项（我们稍后讲解）。

(3)　服务器必须确认（ACK）客户的SYN，同时自己也得发送一个SYN分节，它含有服务器将在同一连接中发送的数据的初始序列号。服务器在单个分节中发送SYN和对客户SYN的ACK（确认）。

(4) 客户必须确认服务器的SYN。

这种交换至少需要3个分组，因此称之为TCP的三路握手（three-way handshake）。图2-2展示了所交换的3个分节。

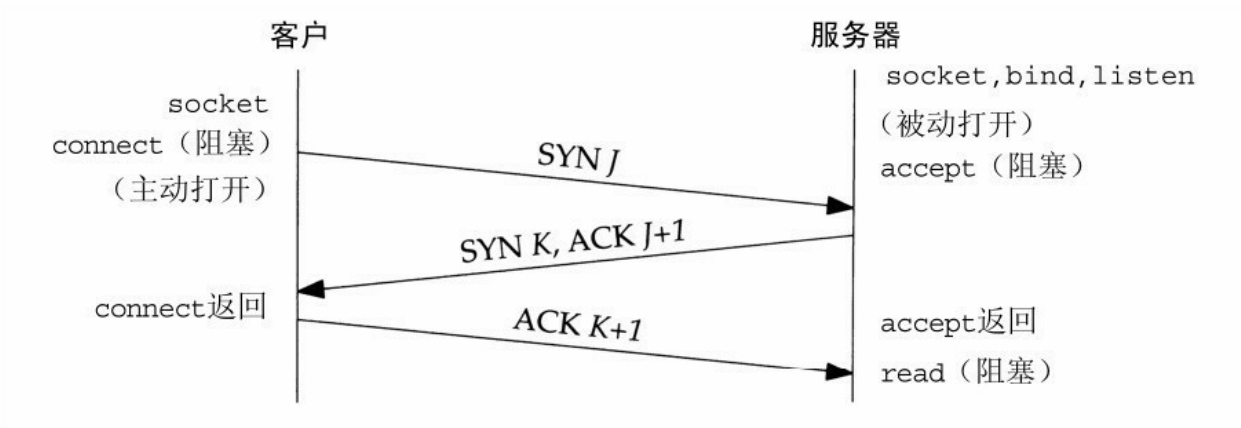


图2-2 TCP的三路握手

图2-2给出的客户的初始序列号为J，服务器的初始序列号为K。ACK中的确认号是发送这个ACK的一端所期待的下一个序列号。因为SYN占据一个字节的序列号空间，所以每一个SYN的ACK中的确认号就是该SYN的初始序列号加1。类似地，每一个FIN（表示结束）的ACK

中的确认号为该FIN的序列号加1。

建立TCP连接就好比一个电话系统［Nemeth 1997］。socket函数等同于有电话可用。bind函数是在告诉别人你的电话号码，这样他们可以呼叫你。listen函数是打开电话振铃，这样当有一个外来呼叫到达时，你就可以听到。connect函数要求我们知道对方的电话号码并拨打它。accept函数发生在被呼叫的人应答电话之时。由accept返回客户的标识（即客户的IP地址和端口号）类似于让电话机的呼叫者ID功能部件显示呼叫者的电话号码。然而两者的不同之处在于accept只在连接建立之后返回客户的标识，而呼叫者ID功能部件却在我们选择应答或不应答电话之前显示呼叫者的电话号码。如果使用域名系统DNS（见第11章），它就提供了一种类似于电话簿的服务。getaddrinfo类似于在电话簿中查找某个人的电话号码，getnameinfo则类似于有一本按照电话号码而不是按照用户名排序的电话簿。

2.6.2 TCP选项

每一个SYN可以含有多个TCP选项。下面是常用的TCP选项。

MSS选项。发送SYN的TCP一端使用本选项通告对端它的最大分节大小（maximum segment size）即MSS，也就是它在本连接的每个TCP分节中愿意接受的最大数据量。发送端TCP使用接收端的MSS值作为所发送分节的最大大小。我们将在7.9节看到如何使用TCP_MAXSEG套接字选项提取和设置这个TCP选项。

窗口规模选项。TCP连接任何一端能够通告对端的最大窗口大小是65535，因为在TCP首部中相应的字段占16位。然而当今因特网上业已普及的高速网络连接（45 Mbit/s或更快，如RFC 1323［Jacobson, Braden, and Borman 1992］所述）或长延迟路径（卫星链路）要求有更大的窗口以获得尽可能大的吞吐量。这个新选项指定TCP首部中的通告窗口必须扩大（即左移）的位数（0～14），因此所提供的最大窗口接

近1 GB（65535×2^{14} ）。在一个TCP连接上使用窗口规模的前提是它的两个端系统必须都支持这个选项。我们将在7.5节看到如何使用SO_RCVBUF套接字选项影响这个TCP选项。

为提供与不支持这个选项的较早实现间的互操作性，需应用如下规则。TCP可以作为主动打开的部分内容随它的SYN发送该选项，但是只在对端也随它的SYN发送该选项的前提下，它才能扩大自己窗口的规模。类似地，服务器的TCP只有接收到随客户的SYN到达的该选项时，才能发送该选项。本逻辑假定实现忽略它们不理解的选项，如此忽略是必需的要求，也已普遍满足，但无法保证所有实现都满足此要求。

时间戳选项。这个选项对于高速网络连接是必要的，它可以防止由失而复现的分组 [11] 可能造成的数据损坏。它是一个较新的选项，也以类似于窗口规模选项的方式协商处理。作为网络编程人员，我们无需考虑这个选项。

TCP的大多数实现都支持这些常用选项。后两个选项有时称为“RFC 1323选项”，因为它们是在RFC 1323［Jacobson, Braden, and Borman 1992］中说明的。既然高带宽或长延迟的网络被称为“长胖管道”（long fat pipe），这两个选项也称为“长胖管道选项”。TCPv1的第24章对这些选项有详细的叙述。

2.6.3 TCP连接终止

TCP建立一个连接需3个分节，终止一个连接则需4个分节。

(1) 某个应用进程首先调用close，我们称该端执行主动关闭（active close）。该端的TCP于是发送一个FIN分节，表示数据发送完毕。

(2) 接收到这个FIN的对端执行被动关闭（passive close）。这个FIN由TCP确认。它的接收也作为一个文件结束符（end-of-file）传递给接收端应用进程（放在已排队等候该应用进程接收的任何其他数据之后），因为FIN的接收意味着接收端应用进程在相应连接上再无额外数据可接

收。

(3) 一段时间后，接收到这个文件结束符的应用进程将调用close关闭它的套接字。这导致它的TCP也发送一个FIN。

(4) 接收这个最终FIN的原发送端TCP（即执行主动关闭的那一端）确认这个FIN。

既然每个方向都需要一个FIN和一个ACK，因此通常需要4个分节。我们使用限定词“通常”是因为：某些情形下步骤1的FIN随数据一起发送；另外，步骤2和步骤3发送的分节都出自执行被动关闭那一端，有可能被合并成一个分节。图2-3展示了这些分组。

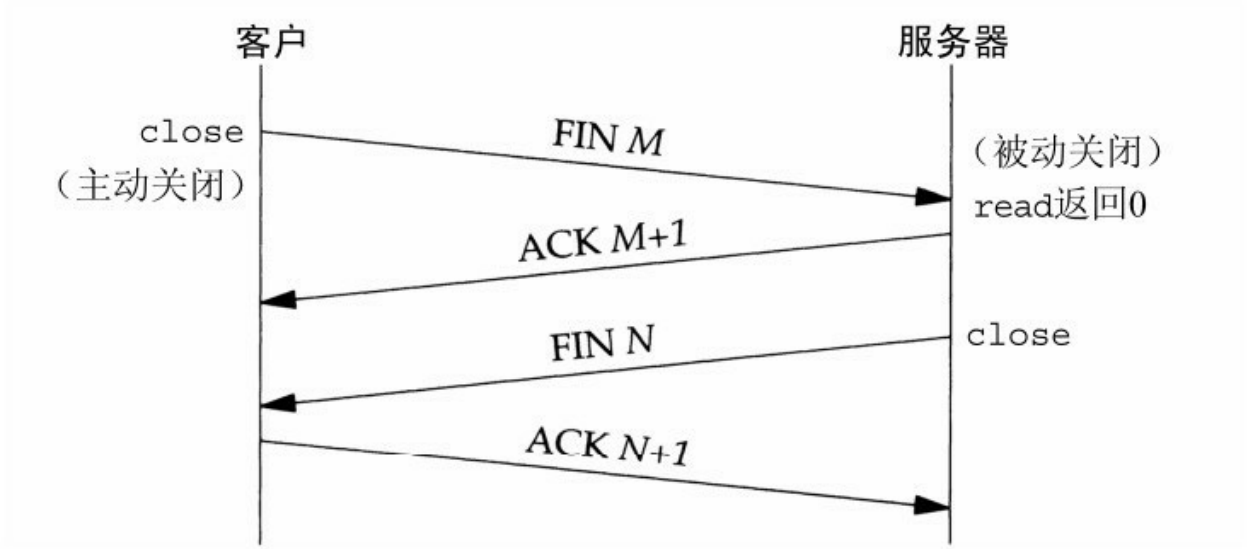


图2-3 TCP连接关闭时的分组交换

类似SYN，一个FIN也占据1个字节的序列号空间。因此，每个FIN的ACK确认号就是这个FIN的序列号加1。

在步骤2与步骤3之间，从执行被动关闭一端到执行主动关闭一端流动数据是可能的。这称为半关闭（half-close），我们将在6.6节随shutdown函数再详细介绍。

当套接字被关闭时，其所在端TCP各自发送了一个FIN。我们在图中指出，这是由应用进程调用close而发生的，不过需认识到，当一个

Unix进程无论自愿地（调用exit或从main函数返回）还是非自愿地（收到一个终止本进程的信号）终止时，所有打开的描述符都被关闭，这也导致仍然打开的任何TCP连接上也发出一个FIN。

图2-3展示了客户执行主动关闭的情形，不过我们指出，无论是客户还是服务器，任何一端都可以执行主动关闭。通常情况是客户执行主动关闭，但是某些协议（譬如值得注意的HTTP/1.0）却由服务器执行主动关闭。

2.6.4 TCP状态转换图

TCP涉及连接建立和连接终止的操作可以用状态转换图（state transition diagram）来说明，如图2-4所示。

TCP为一个连接定义了11种状态，并且TCP规则规定如何基于当前状态及在该状态下所接收的分节从一个状态转换到另一个状态。举例来说，当某个应用进程在CLOSED状态下执行主动打开时，TCP将发送一个SYN，且新的状态是SYN_SENT。如果这个TCP接着接收到一个带ACK的SYN，它将发送一个ACK，且新的状态是ESTABLISHED。这个最终状态是绝大多数数据传送发生的状态。

自ESTABLISHED状态引出的两个箭头处理连接的终止。如果某个应用进程在接收到一个FIN之前调用close（主动关闭），那就转换到FIN_WAIT_1状态。但如果某个应用进程在ESTABLISHED状态期间接收到一个FIN（被动关闭），那就转换到CLOSE_WAIT状态。

我们用粗实线表示通常的客户状态转换，用粗虚线表示通常的服务器状态转换。图中还注明存在两个我们未曾讨论的转换：一个为同时打开（simultaneous open），发生在两端几乎同时发送SYN并且这两个SYN在网络中交错的情形下，另一个为同时关闭（simultaneous close），发生在两端几乎同时发送FIN的情形下。TCPv1的第18章中有这两种情况的例子和讨论，它们是可能发生的，不过非常罕见。

展示状态转换图的原因之一是给出11种TCP状态的名称。这些状态

可使用netstat显示，它是一个在调试客户/服务器应用时很有用的工具。我们将在第5章中使用netstat去监视状态的变化。

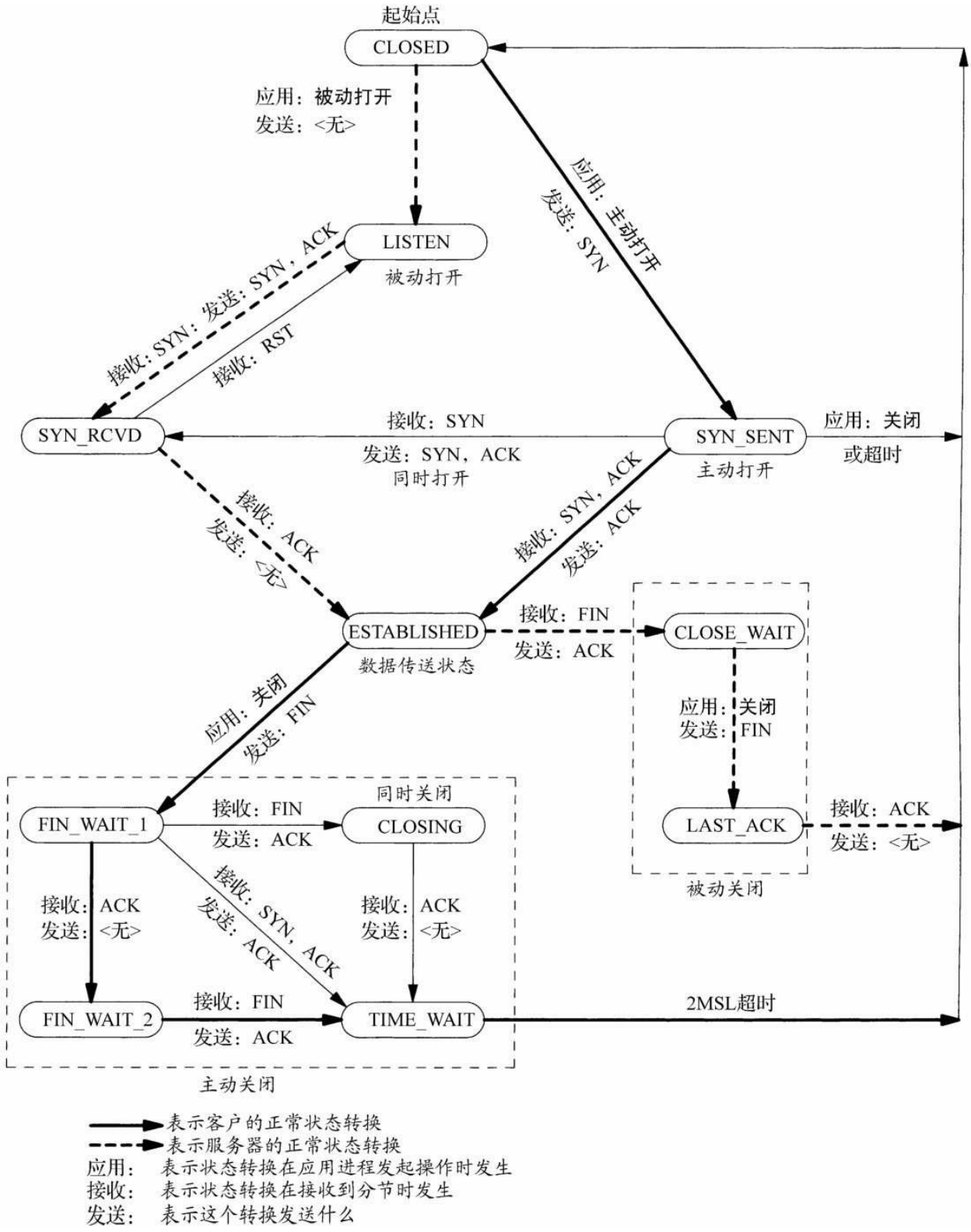


图2-4 TCP状态转换图

2.6.5 观察分组

图2-5展示一个完整的TCP连接所发生的实际分组交换情况，包括连接建立、数据传送和连接终止3个阶段。图中还展示了每个端点所历经的TCP状态。

本例中的客户通告一个值为536的MSS（表明该客户只实现了最小重组缓冲区大小），服务器通告一个值为1460的MSS（以太网上IPv4的典型值）。不同方向上MSS值不相同不成问题（见习题2.5）。

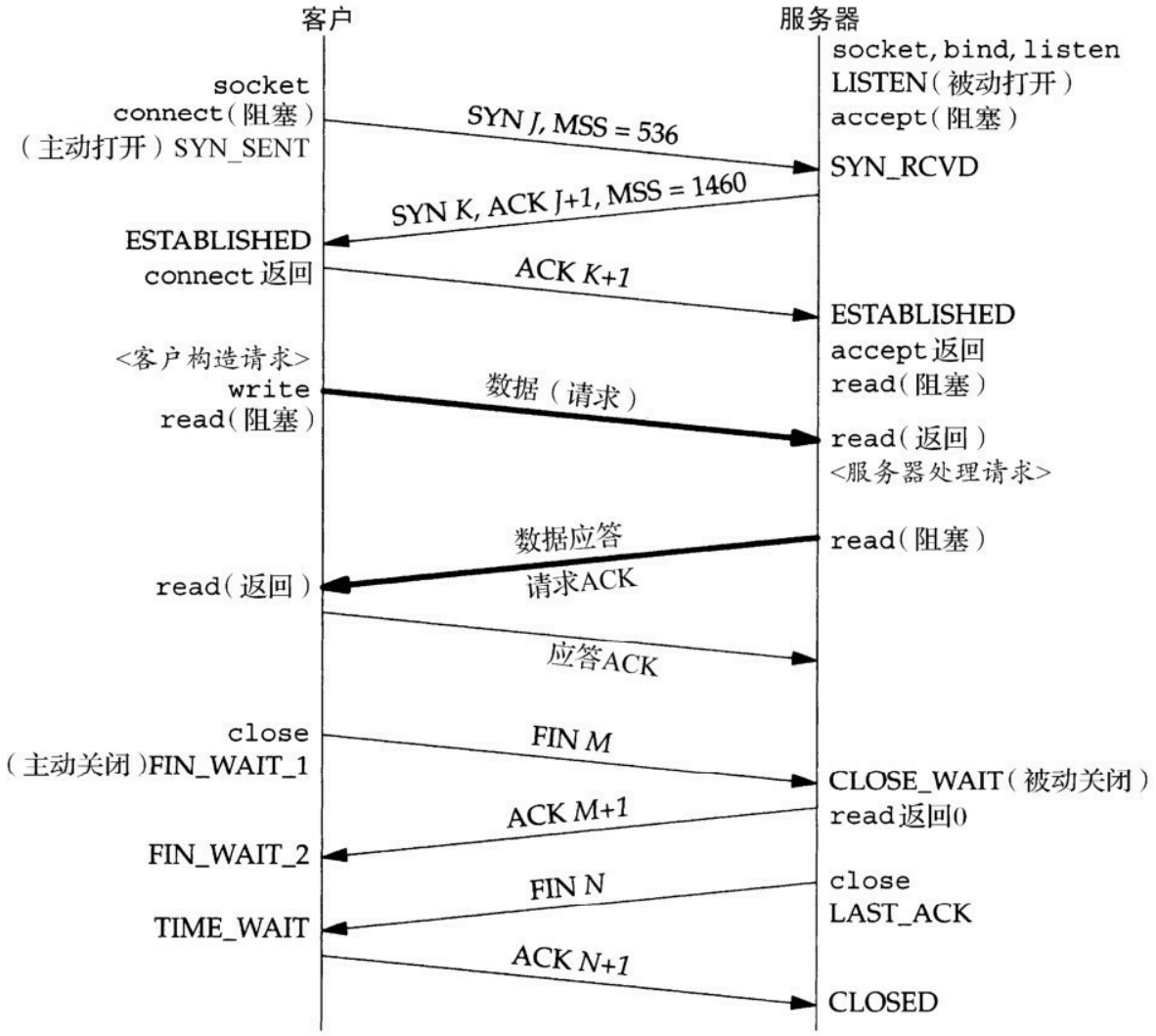


图2-5 TCP连接的分组交换

一旦建立一个连接，客户就构造一个请求并发送给服务器。这里我们假设该请求适合于单个TCP分节（即请求大小小于服务器通告的值为1460字节的MSS）。服务器处理该请求并发送一个应答，我们假设该应答也适合于单个分节（本例即小于536字节）。图中使用粗箭头表示这两个数据分节。注意，服务器对客户请求的确认是伴随其应答发送的。这种做法称为捎带（piggybacking），它通常在服务器处理请求并产生应答的时间少于200 ms时发生。如果服务器耗用更长时间，譬如说1 s，那么我们将看到先是确认后是应答。（TCP数据流机理在TCPv1的第19

章和第20章中详细叙述。）

图中随后展示的是终止连接的4个分节。注意，执行主动关闭的那一端（本例子中为客户）进入我们将在下一节中讨论的TIME_WAIT状态。

图2-5中值得注意的是，如果该连接的整个目的仅仅是发送一个单分节的请求和接收一个单分节的应答，那么使用TCP有8个分节的开销。如果改用UDP，那么只需交换两个分组：一个承载请求，一个承载应答。然而从TCP切换到UDP将丧失TCP提供给应用进程的全部可靠性，迫使可靠服务的一大堆细节从传输层（TCP）转移到UDP应用进程。TCP提供的另一个重要特性即拥塞控制也必须由UDP应用进程来处理。尽管如此，我们仍然需要知道许多网络应用是使用UDP构建的，因为它们需要交换的数据量较少，而UDP避免了TCP连接建立和终止所需的开销。

2.7 TIME_WAIT状态

毫无疑问，TCP中有关网络编程最不容易理解的是它的TIME_WAIT状态。在图2-4中我们看到执行主动关闭的那端经历了这个状态。该端点停留在这个状态的持续时间是最长分节生命期（maximum segment lifetime，MSL）的两倍，有时候称之为2MSL。

任何TCP实现都必须为MSL选择一个值。RFC 1122［Braden 1989］的建议值是2分钟，不过源自Berkeley的实现传统上改用30秒这个值。这意味着TIME_WAIT状态的持续时间在1分钟到4分钟之间。MSL是任何IP数据报能够在因特网中存活的最长时间。我们知道这个时间是有限的，因为每个数据报含有一个称为跳限（hop limit）的8位字段（见图A-1中IPv4的TTL字段和图A-2中IPv6的跳限字段），它的最大值为255。尽管这是一个跳数限制而不是真正的时间限制，我们仍然假设：具有最大

跳限（255）的分组在网络中存在的时间不可能超过MSL秒。

分组在网络中“迷途”通常是路由异常的结果。某个路由器崩溃或某两个路由器之间的某个链路断开时，路由协议需花数秒钟到数分钟的时间才能稳定并找出另一条通路。在这段时间内有可能发生路由循环（路由器A把分组发送给路由器B，而B再把它们发送回A），我们关心的分组可能就此陷入这样的循环。假设迷途的分组是一个TCP分节，在它迷途期间，发送端TCP超时并重传该分组，而重传的分组却通过某条候选路径到达最终目的地。然而不久后（自迷途的分组开始其旅程起最多MSL秒以内）路由循环修复，早先迷失在这个循环中的分组最终也被送到目的地。这个原来的分组称为迷途的重复分组（lost duplicate）或漫游的重复分组（wandering duplicate）。TCP必须正确处理这些重复的分组。

TIME_WAIT状态有两个存在的理由：

(1) 可靠地实现TCP全双工连接的终止；

(2) 允许老的重复分节在网络中消逝。

第一个理由可以通过查看图2-5并假设最终的ACK丢失了来解释。服务器将重新发送它的最终那个FIN，因此客户必须维护状态信息，以允许它重新发送最终那个ACK。要是客户不维护状态信息，它将响应以一个RST（另外一种类型的TCP分节），该分节将被服务器解释成一个错误。如果TCP打算执行所有必要的工作以彻底终止某个连接上两个方向的数据流（即全双工关闭），那么它必须正确处理连接终止序列4个分节中任何一个分节丢失的情况。本例子也说明了为什么执行主动关闭的那一端是处于TIME_WAIT状态的那一端：因为可能不得不重传最终那个ACK的就是那一端。

为理解存在TIME_WAIT状态的第二个理由，我们假设在12.106.32.254的1500端口和206.168.112.219的21端口之间有一个TCP连接。我们关闭这个连接，过一段时间后在相同的IP地址和端口之间建立

另一个连接。后一个连接称为前一个连接的化身（incarnation），因为它们的IP地址和端口号都相同。TCP必须防止来自某个连接的老的重复分组在该连接已终止后再现，从而被误解成属于同一连接的某个新的化身。为做到这一点，TCP将不给处于TIME_WAIT状态的连接发起新的化身。既然TIME_WAIT状态的持续时间是MSL的2倍，这就足以让某个方向上的分组最多存活MSL秒即被丢弃，另一个方向上的应答最多存活MSL秒也被丢弃。通过实施这个规则，我们就能保证每成功建立一个TCP连接时，来自该连接先前化身的老的重复分组都已在网络中消逝了。

这个规则存在一个例外：如果到达的SYN的序列号大于前一化身的结束序列号，源自Berkeley的实现将给当前处于TIME_WAIT状态的连接启动新的化身。TCPv2第958～959页对这种情况有详细的叙述。它要求服务器执行主动关闭，因为接收下一个SYN的那一端必须处于TIME_WAIT状态。rsh命令具备这种能力。RFC 1185［Jacobson, Braden, and Zhang 1990］讲述了有关这种情形的一些陷阱。

2.8 SCTP关联的建立和终止

与TCP一样，SCTP也是面向连接的，因而也有关联的建立与终止的握手过程。不过SCTP的握手过程不同于TCP，我们在此加以说明。

2.8.1 四路握手

建立一个SCTP关联的时候会发生下述情形（类似于TCP）。

(1) 服务器必须准备好接受外来的关联。这通常通过调用socket、bind和listen这3个函数来完成，称为被动打开。

(2) 客户通过调用connect或者发送一个隐式打开该关联的消息进行主动打开。这使得客户SCTP发送一个INIT消息（初始化），该消息告诉服务器客户的IP地址清单、初始序列号、用于标识本关联中所有分组

的起始标记、客户请求的外出流的数目以及客户能够支持的外来流的数目。

(3) 服务器以一个INIT ACK消息确认客户的INIT消息，其中含有服务器的IP地址清单、初始序列号、起始标记、服务器请求的外出流的数目、服务器能够支持的外来流的数目以及一个状态cookie。状态cookie包含服务器用于确信本关联有效所需的所有状态，它是数字化签名过的，以确保其有效性。

(4) 客户以一个COOKIE ECHO消息回射服务器的状态cookie。除COOKIE ECHO外，该消息可能在同一个分组中还捆绑了用户数据。

(5) 服务器以一个COOKIE ACK消息确认客户回射的cookie是正确的，本关联于是建立。该消息也可能在同一个分组中还捆绑了用户数据。

以上交换过程至少需要4个分组，因此称之为SCTP的四路握手（four-way handshake）。图2-6展示了这4个分节。

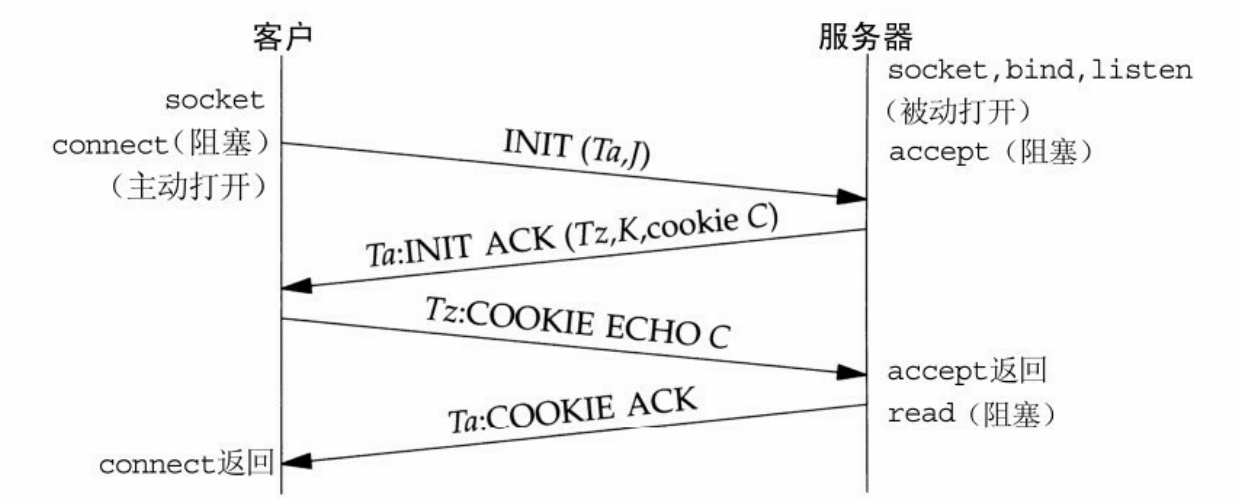


图2-6 SCTP的四路握手

SCTP的四路握手在很多方面类似于TCP的三路握手，差别主要在于作为SCTP整体一部分的cookie的生成。INIT（随其众多参数一道）承载一个验证标记Ta和一个初始序列号J。在关联的有效期内，验证标记

Ta必须在对端发送的每个分组中出现。初始序列号J用作承载用户数据的DATA块的起始序列号。对端也在INIT ACK中承载一个验证标记Tz，在关联的有效期内，验证标记Tz也必须在其发送的每个分组中出现。除了验证标记Tz和初始序列号K外，INIT的接收端还在作为响应的INIT ACK中提供一个cookie C。该cookie包含设置本SCTP关联所需的所有状态，这样服务器的SCTP栈就不必保存所关联客户的有关信息。SCTP关联设置的细节参见［Stewart and Xie 2001］的第4章。

四路握手过程结束时，两端各自选择一个主目的地址（primary destination address）。当不存在网络故障时，主目的地址将用作数据要发送到的默认目的地。

在SCTP中使用四路握手是为了避免一种将在4.5节讨论的拒绝服务攻击。

SCTP使用cookie的四路握手定形了一种防护这种攻击的方法。TCP的许多实现也使用类似的方法。两者的主要差别在于，TCP中cookie状态必须编码到只有32位长的初始序列号中。SCTP为此提供了一个任意长度的字段，并且要求实施基于加密的安全性以防护攻击。

2.8.2 关联终止

SCTP不像TCP那样允许“半关闭”的关联。当一端关闭某个关联时，另一端必须停止发送新的数据。关联关闭请求的接收端发送完已经排队的数据（如果有的话）后，完成关联的关闭。图2-7展示了这一交换过程。

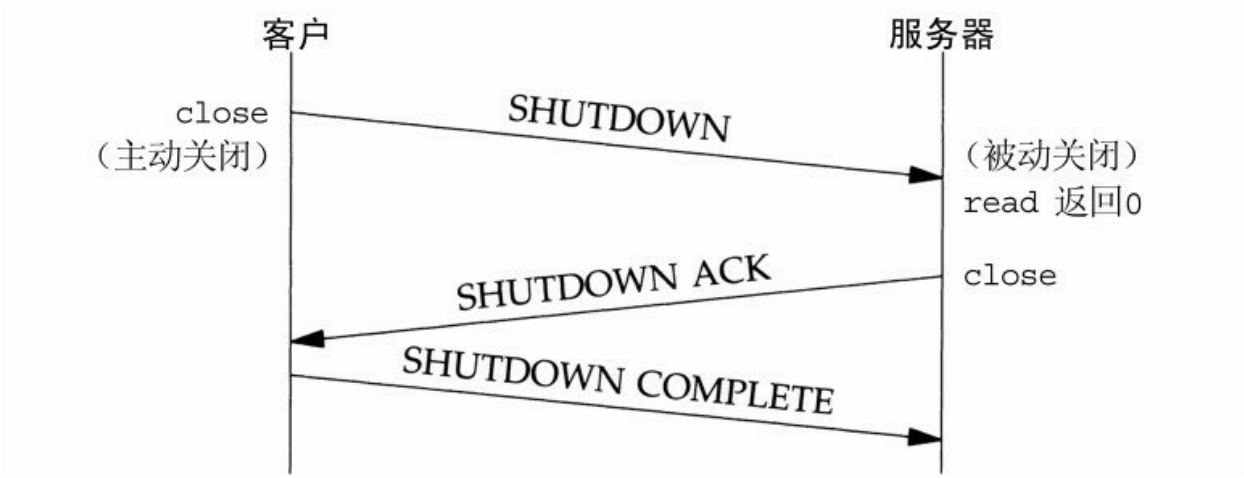


图2-7 SCTP关联关闭时的分组交换

SCTP没有类似于TCP的TIME_WAIT状态，因为SCTP使用了验证标记。所有后续块都在捆绑它们的SCTP分组的公共首部标记了初始的INIT块和INIT ACK块中作为起始标记交换的验证标记；由来自旧连接的块通过所在SCTP分组的公共首部间接携带的验证标记对于新连接来说是不正确的。因此，SCTP通过放置验证标记值就避免了TCP在TIME_WAIT状态保持整个连接的做法。

2.8.3 SCTP状态转换图

SCTP涉及关联建立和关联终止的操作可以用状态转换图（state transition diagram）来说明，如图2-8所示。

与图2-4一样，本状态机中从一个状态到另一个状态的转换由SCTP规则基于当前状态及在该状态下所接收的块规定。举例来说，当某个应用进程在CLOSED状态下执行主动打开时，SCTP将发送一个INIT，且新的状态是COOKIE-WAIT。如果这个SCTP接着接收到一个INIT ACK，它将发送一个COOKIE ECHO，且新的状态是COOKIE-ECHOED。如果该SCTP随后接收到一个COOKIE ACK，它将转换成ESTABLISHED状态。这个最终状态是绝大多数数据传送发生点的状态，尽管DATA块也可以由COOKIE ECHO块或COOKIE ACK块所在消息捆绑捎带。

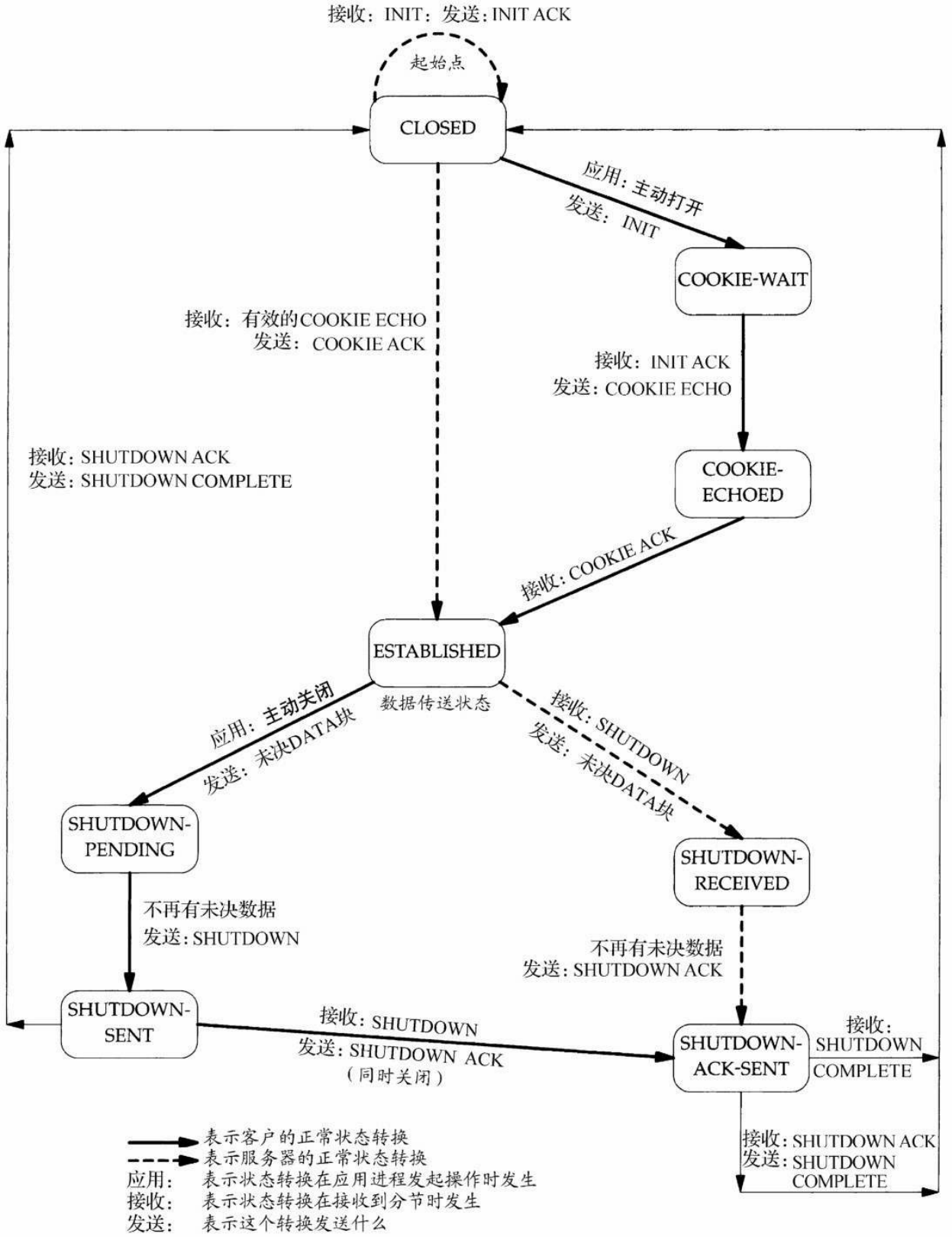


图2-8 SCTP状态转换图

从ESTABLISHED状态引出的两个箭头处理关联的终止。如果某个应用进程在接收到一个SHUTDOWN之前调用close（主动关闭），那就转换到SHUTDOWN-PENDING状态。否则，如果某个应用进程在ESTABLISHED状态期间接收到一个SHUTDOWN（被动关闭），那就转换到SHUTDOWN-RECEIVED状态。

2.8.4 观察分组

图2-9展示一个作为样例的SCTP关联所发生的实际分组交换情况，包括关联建立、数据传送和关联终止3个阶段。图中还展示了每个端点所历经的SCTP状态。

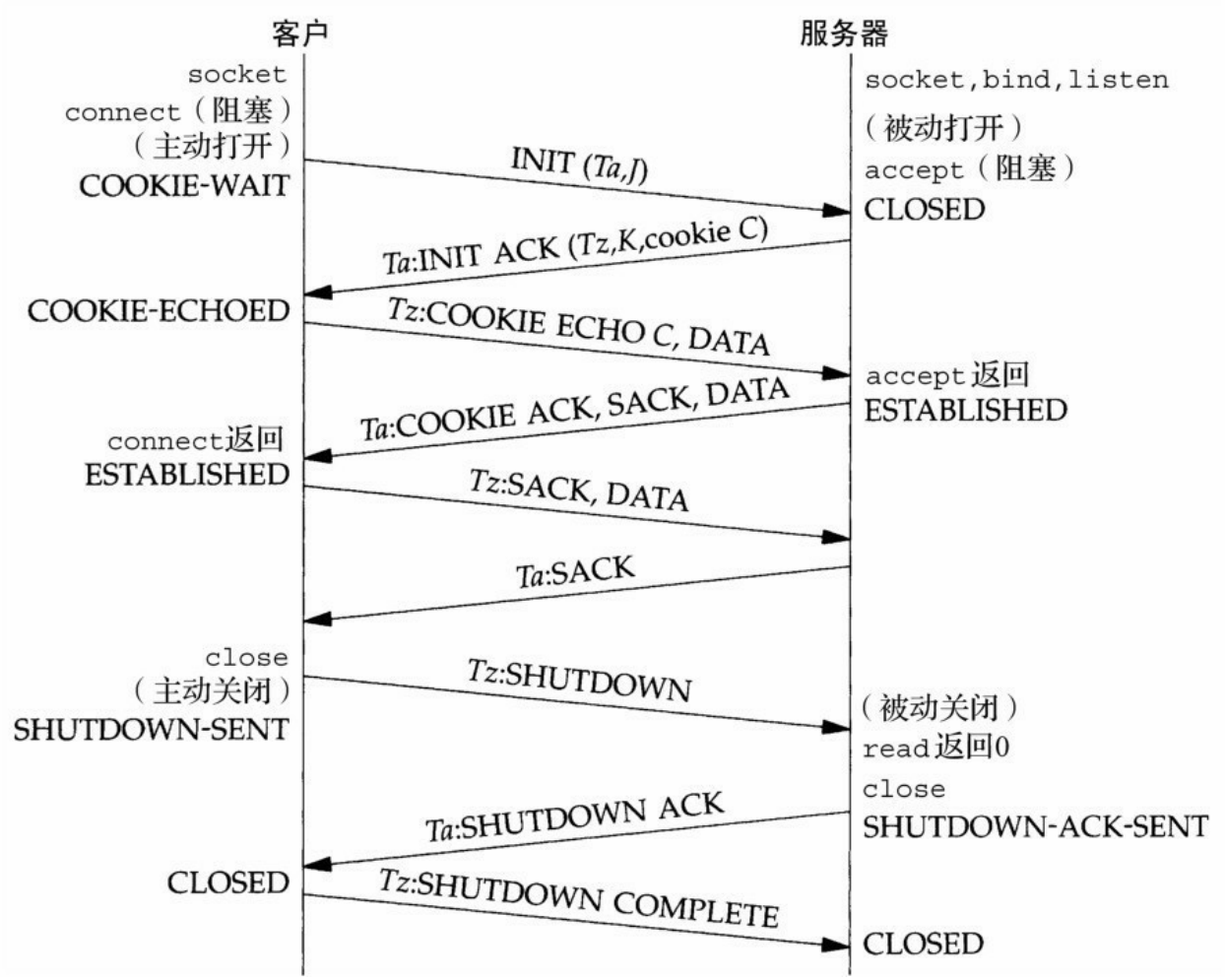


图2-9 SCTP关联中的分组交换

本例中，客户在COOKIE　ECHO块所在分组中捎带了它的第一个DATA块，服务器则在作为应答的COOKIE ACK块所在分组中捎带了数据。一般而言，当网络应用采用一到多接口式样时（我们将在9.2节中讨论一到一和一到多这两种接口式样），COOKIE ECHO通常捎带一个或多个DATA块。

SCTP分组中信息的单位称为块（chunk）。块是自描述的，包含一个块类型、若干个块标记和一个块长度。这样做方便了多个块的绑缚，只要把它们简单地组合到一个SCTP外出消息中（［Stewart and Xie 2001］的第5章给出了块捆绑和常规数据传输过程的细节）。

2.8.5 SCTP选项

SCTP使用参数和块方便增设可选特性。新的特性通过添加这两个条目之一加以定义，并允许通常的SCTP处理规则汇报未知的参数和未知的块。参数类型字段和块类型字段的高两位指明SCTP接收端该如何处置未知的参数或未知的块（［Stewart and Xie 2001］的3.1节给出了更多的细节）。

当前如下两个对SCTP的扩展正在开发中。

(1) 动态地址扩展，允许协作的SCTP端点从已有的某个关联中动态增删IP地址。

(2) 不完全可靠性扩展，允许协作的SCTP端点在应用进程的指导下限制数据的重传。当一个消息变得过于陈旧而无须发送时（按照应用进程的指导），该消息将被跳过而不再发送到对端。这意味着不是所有数据都确保到达关联的另一端。

2.9 端口号 ■

任何时候，多个进程可能同时使用TCP、UDP和SCTP这3种传输层协议中的任何一种。这3种协议都使用16位整数的端口号（port

number）来区分这些进程。

当一个客户想要跟一个服务器联系时，它必须标识想要与之通信的这个服务器。TCP、UDP和SCTP定义了一组众所周知的端口（well-known port），用于标识众所周知的服务。举例来说，支持FTP的任何TCP/IP实现都把21这个众所周知的端口分配给FTP服务器。分配给简化文件传送协议（Trivial File Trqnsfer Protocol，TFTP）的是UDP端口号69。

另一方面，客户通常使用短期存活的临时端口（ephemeral port）。这些端口号通常由传输层协议自动赋予客户。客户通常不关心其临时端口的具体值，而只需确信该端口在所在主机中是唯一的就行。传输协议的代码确保这种唯一性。

IANA（the Internet Assigned Numbers Authority，因特网已分配数值权威机构）维护着一个端口号分配状况的清单。该清单一度作为RFC多次发布；RFC 1700［Reynolds and Postel 1994］是这个系列的最后一个。RFC 3232［Reynolds 2002］给出了替代RFC 1700的在线数据库的位置：http://www.iana.org/。端口号被划分成以下3段。

(1) 众所周知的端口为0～1023。这些端口由IANA分配和控制。可能的话，相同端口号就分配给TCP、UDP和SCTP的同一给定服务。例如，不论TCP还是UDP端口号80都被赋予Web服务器，尽管它目前的所有实现都单纯使用TCP。

端口号80分配时SCTP尚不存在。新的端口分配将针对这3种协议执行，RFC 2960则声明所有现有的TCP端口号对于使用SCTP的同一服务同样有效。

(2) 已登记的端口（registered port）为1024～49151。这些端口不受IANA控制，不过由IANA登记并提供它们的使用情况清单，以方便整个群体。可能的话，相同端口号也分配给TCP和UDP的同一给定服务。例如，6000～6063分配给这两种协议的X Window服务器，尽管它的所有

实现当前单纯使用TCP。49151这个上限的引入是为了给临时端口留出范围，而RFC 1700［Reynolds and Postel 1994］所列的上限为65535。

(3) 49152～65535是动态的（dynamic）或私用的（private）端口。IANA不管这些端口。它们就是我们所称的临时端口。（49152这个魔数是65536的四分之三。）

图2-10展示了端口号的划分情况和常见的分配情况。

我们要注意图2-10中以下几点。

Unix系统有保留端口（reserved port）的概念，指的是小于1024的任何端口。这些端口只能赋予特权用户进程的套接字。所有IANA众所周知的端口都是保留端口，分配使用这些端口的服务器（例如FTP服务器）必须以超级用户特权启动。

由于历史原因，源自Berkeley的实现（从4.3BSD开始）曾在1024～5000范围内分配临时端口。这在20世纪80年代初期是可行的，但是如今很容易就找到一个在任何给定时间内同时支持多于3977个连接的主机。于是许多较新的系统从另外的范围分配临时端口以提供更多的临时端口，它们或者使用由IANA定义的临时端口范围，或者使用一个更大的其他范围（如图2-10所示的Solaris）。

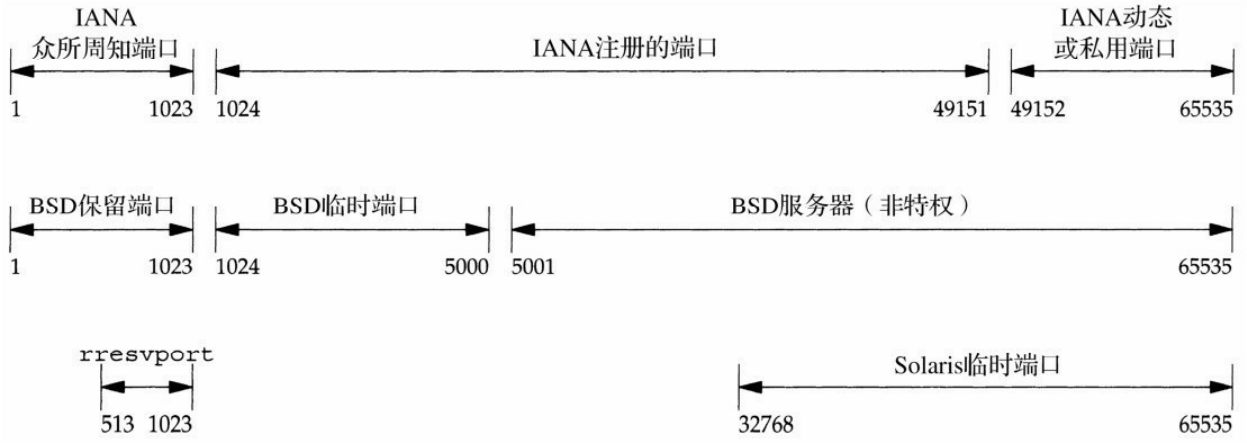


图2-10 端口号的分配

由于这个原因，许多较早的系统实现的临时端口范围的上限为5

000。5 000这个上限后来发现是一个排版错误［Borman 1997a］，本应该是50 000。

有少数客户（而不是服务器）需要一个保留端口用于客户/服务器的认证：rlogin和rsh客户就是常见的例子。这些客户调用库函数rresvport创建一个TCP套接字，并赋予它一个在513～1023范围内未使用的端口。该函数通常先尝试绑定端口1023，若失败则尝试1022，依次类推，直到在端口513上亦或成功，亦或失败。

注意：BSD的保留端口和rresvport函数都跟IANA众所周知端口的后半部分重叠。这是因为IANA众所周知端口早先的上限为255。1992年的RFC 1340（早先的一个“Assigned Numbers”RFC）开始在256～1023之间分配众所周知的端口。1990年的RFC 1060（更早先的一个“Assigned Numbers”RFC）称256～1023之间的端口为Unix标准服务（Unix Standard Services）。20世纪80年代有不少源自Berkeley的服务器在512以后挑选它们的众所周知的端口（留下256～511这个空档）。rresvport函数选择从1023开始往下寻找，直至513。

套接字对

一个TCP连接的套接字对（socket pair）是一个定义该连接的两个端点的四元组：本地IP地址、本地TCP端口号、外地IP地址、外地TCP端口号。套接字对唯一标识一个网络上的每个TCP连接。就SCTP而言，一个关联由一组本地IP地址、一个本地端口、一组外地IP地址、一个外地端口标识。在两个端点均非多宿这一最简单的情形下，SCTP与TCP所用的四元组套接字对一致。然而在某个关联的任何一个端点为多宿的情形下，同一个关联可能需要多个四元组标识（这些四元组的IP地址各不相同，但端口号是一样的）。

标识每个端点的两个值（IP地址和端口号）通常称为一个套接字。

我们可以把套接字对的概念扩展到UDP，即使UDP是无连接的。当讲解套接字函数（bind、connect、getpeername等）时，我们将指明它们

在指定套接字对中的哪些值。举例来说，bind函数要求应用程序给TCP、UDP或SCTP套接字指定本地IP地址和本地端口号。

2.10 TCP端口号与并发服务器

并发服务器中主服务器循环通过派生一个子进程来处理每个新的连接。如果一个子进程继续使用服务器众所周知的端口来服务一个长时间的请求，那将发生什么？让我们来看一个典型的序列。首先，在主机freebsd上启动服务器，该主机是多宿的，其IP地址为12.106.32.254和192.168.42.1。服务器在它的众所周知的端口（本例为21）上执行被动打开，从而开始等待客户的请求，如图2-11所示。

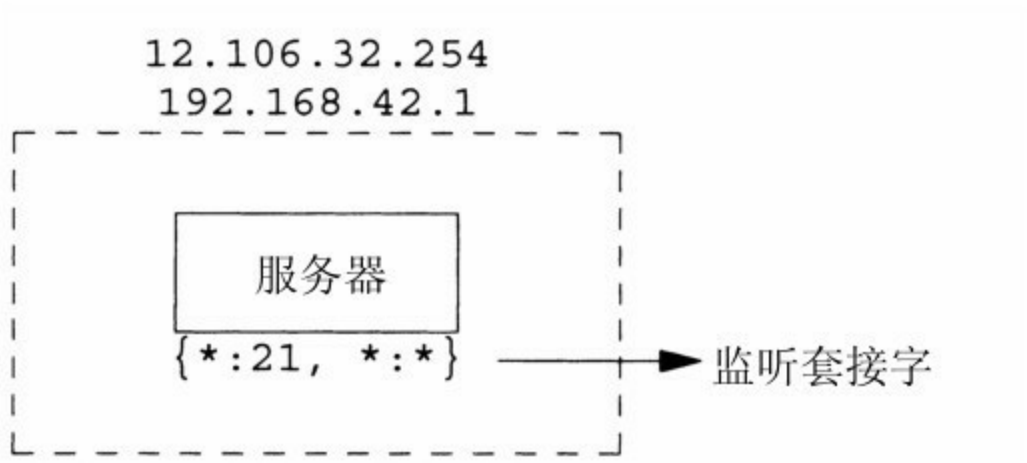


图2-11 TCP服务器在端口21上执行被动打开

我们使用记号{*:21, *:*}指出服务器的套接字对。服务器在任意本地接口（第一个星号）的端口21上等待连接请求。外地IP地址和外地端口都没有指定，我们用“*.*”来表示。我们称它为监听套接字（listening socket）。

我们用分号来分割IP地址和端口号，因为这是HTTP的用法，其他地方也常见。netstat程序使用点号来分割IP地址和端口号，不过如此表示有时候会让人混淆，因为点号既用于域名（如

freebsd.unpbook.com.21），也用于IPv4的点分十进制数记法（如12.106.32. 254.21）。

这里指定本地IP地址的星号称为通配（wildcard）符。如果运行服务器的主机是多宿的（如本例），服务器可以指定它只接受到达某个特定本地接口的外来连接。这里要么选一个接口要么选任意接口。服务器不能指定一个包含多个地址的清单。通配的本地地址表示“任意”这个选择。在图1-9中，通配地址通过在调用bind之前把套接字地址结构中的IP地址字段设置成INADDR_ANY来指定。

稍后在IP地址为206.168.112.219的主机上启动第一个客户，它对服务器的IP地址之一12.106.32.254执行主动打开。我们假设本例中客户主机的TCP为此选择的临时端口为1500，如图2-12所示。图中在该客户的下方标出了它的套接字对。

图2-12 客户对服务器的连接请求

当服务器接收并接受这个客户的连接时，它fork一个自身的副本，让子进程来处理该客户的请求，如图2-13所示。（我们将在4.7节中讲解fork函数。）

至此，我们必须在服务器主机上区分监听套接字和已连接套接字（connected socket）。注意已连接套接字使用与监听套接字相同的本地端口（21）。还要注意在多宿服务器主机上，连接一旦建立，已连接套接字的本地地址（12.106.32.254）随即填入。

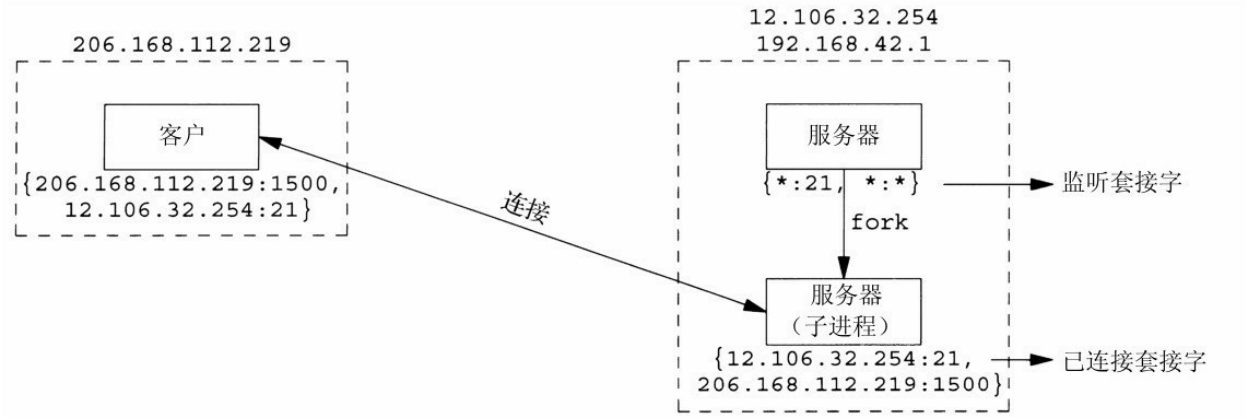


图2-13 并发服务器让子进程处理客户

下一步我们假设在客户主机上另有一个客户请求连接到同一个服务器。客户主机的TCP为这个新客户的套接字分配一个未使用的临时端口，譬如说1501，如图2-14所示。服务器上这两个连接是有区别的：第一个连接的套接字对和第二个连接的套接字对不一样，因为客户的TCP给第二个连接选择了一个未使用的端口（1501）。

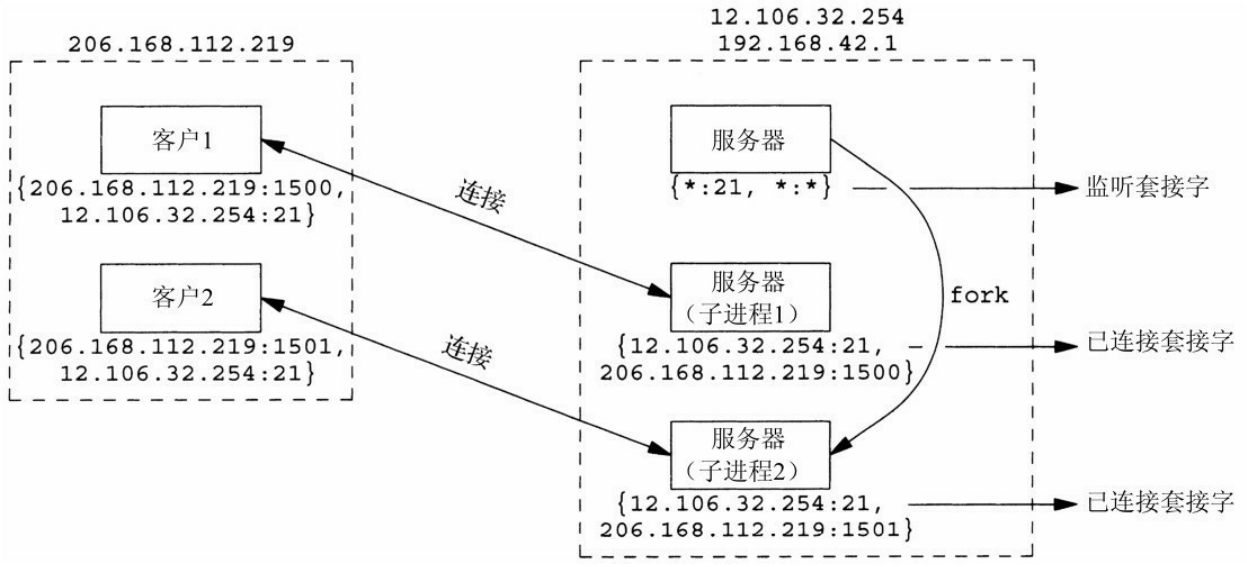


图2-14 第二个客户与同一个服务器的连接

通过本例应注意，TCP无法仅仅通过查看目的端口号来分离外来的分节到不同的端点。它必须查看套接字对的所有4个元素才能确定由哪个端点接收某个到达的分节。图2-14中对于同一个本地端口（21）存在

3个套接字。如果一个分节来自206.168.112.219端口1500，目的地为12.106.32.254端口21，它就被递送给第一个子进程。如果一个分节来自206.168.112.219端口1501，目的地为12.106.32.254端口21，它就被递送给第二个子进程。所有目的端口为21的其他TCP分节都被递送给拥有监听套接字的最初那个服务器（父进程）。

2.11 缓冲区大小及限制

下面我们将介绍一些影响IP数据报大小的限制。我们首先介绍这些限制，然后就它们如何影响应用进程能够传送的数据进行综合分析。

IPv4数据报的最大大小是65 535字节，包括IPv4首部。这是因为如图A-1所示其总长度字段占据16位。

IPv6数据报的最大大小是65 575字节，包括40字节的IPv6首部。这是因为如图A-2所示其净荷长度字段占据16位。注意，IPv6的净荷长度字段不包括IPv6首部，而IPv4的总长度字段包括IPv4首部。

IPv6有一个特大净荷（jumbo payload）选项，它把净荷长度字段扩展到32位，不过这个选项需要MTU（maximum transmission unit，最大传输单元）超过65 535的数据链路提供支持。（这是为主机到主机的内部连接而设计的，譬如HIPPI，它们通常没有内在的MTU。）

许多网络有一个可由硬件规定的MTU。举例来说，以太网的MTU是1500字节。另有一些链路（例如使用PPP协议的点到点链路）其MTU可以人为配置。较老的SLIP链路通常使用1006字节或296字节的MTU。

IPv4要求的最小链路MTU是68字节。这允许最大的IPv4首部（包括20字节的固定长度部分和最多40字节的选项部分）拼接最小的片段（IPv4首部中片段偏移字段以8个字节为单位）。IPv6要求的最小链路MTU为1280字节。IPv6可以运行在MTU小于此最小值的链路上，不过需要特定于链路的分片和重组功能，以使得这些链路看起来具有至少为

1280字节的MTU（RFC 2460［Deering and Hinden 1998］）。

在两个主机之间的路径中最小的MTU称为路径MTU（path MTU）。1500字节的以太网MTU是当今常见的路径MTU。两个主机之间相反的两个方向上路径MTU可以不一致，因为在因特网中路由选择往往是不对称的［Paxson 1196］，也就是说从A到B的路径与从B到A的路径可以不相同。

当一个IP数据报将从某个接口送出时，如果它的大小超过相应链路的MTU，IPv4和IPv6都将执行分片（fragmentation）。这些片段在到达最终目的地之前通常不会被重组（reassembling）。IPv4主机对其产生的数据报执行分片，IPv4路由器则对其转发的数据报执行分片。然而IPv6只有主机对其产生的数据报执行分片，IPv6路由器不对其转发的数据报执行分片。

我们必须小心这些术语的使用。一个标记为IPv6路由器的设备可能执行分片，不过只是对于那些由它产生的数据报，而绝不是对于那些由它转发的数据报。当该设备产生IPv6数据报时，它实际上作为主机运作。举例来说，大多数路由器支持Telnet协议，管理员就用它来配置路由器。由路由器的Telnet服务器产生的IP数据报是由路由器产生的，而不是由路由器转发的。

你可能注意到，IPv4首部（图A-1）有用于处理IPv4分片的字段，IPv6首部（图A-2）却没有类似的字段。既然分片是例外情况而不是通常情况，IPv6于是引入一个可选首部以提供分片信息。

某些通常用作路由器的防火墙可能会重组分片了的分组，以便查看整个IP数据报的内容。这样做使得不必在防火墙上引入额外的复杂性就能够防止某些攻击。它还要求防火墙设备是进出网络的唯一路径上的设备，从而减少了冗余的机会。

IPv4首部（图A-1）的“不分片（don’t fragment）”位（即DF位）若被设置，那么不管是发送这些数据报的主机还是转发它们的路由器，都

不允许对它们分片。当路由器接收到一个超过其外出链路MTU大小且设置了DF位的IPv4数据报时，它将产生一个ICMPv4“destination unreachable, fragmentation needed but DF bit set”（目的地不可达，需分片但DF位已设置）出错消息（图A-15）。

既然IPv6路由器不执行分片，每个IPv6数据报于是隐含一个DF位。当IPv6路由器接收到一个超过其外出链路MTU大小的IPv6数据报时，它将产生一个ICMPv6“packet too big”（分组太大）出错消息（图A-16）。

IPv4的DF位和IPv6的隐含DF位可用于路径MTU发现（IPv4的情形见RFC 1191［Mogul and Deering 1990］，IPv6的情形见RFC 1981［McCann, Deering, and Mogul 1996］）。举例来说，如果基于IPv4的TCP使用该技术，那么它将在所发送的所有数据报中设置DF位。如果某个中间路由器返回一个ICMP“destination unreachable, fragmentation needed but DF bit set”错误，TCP就减小每个数据报的数据量并重传。路径MTU发现对于IPv4是可选的，然而IPv6的所有实现要么必须支持它，要么必须总是使用最小的MTU发送IPv6数据报。

路径MTU发现在如今的因特网上是有问题的，许多防火墙丢弃所有ICMP消息，包括用于路径MTU发现的上述消息。这意味着TCP永远得不到要求它降低所发送数据量的信号。编写本书时，IETF已经开始尝试定义不依赖于ICMP出错消息的另一种路径MTU发现方法。

IPv4和IPv6都定义了最小重组缓冲区大小（minimum reassembly buffer size），它是IPv4或IPv6的任何实现都必须保证支持的最小数据报大小。其值对于IPv4为576字节，对于IPv6为1500字节。例如，就IPv4而言，我们不能判定某个给定目的地能否接受577字节的数据报。为此有许多使用UDP的IPv4网络应用（如DNS、RIP、TFTP、BOOTP、SNMP）避免产生大于这个大小的数据报。

TCP有一个MSS（maximum segment size，最大分节大小），用于向对端TCP通告对端在每个分节中能发送的最大TCP数据量。在图2-5中

我们看到过SYN分节上的MSS选项。MSS的目的是告诉对端其重组缓冲区大小的实际值，从而试图避免分片。MSS经常设置成MTU减去IP和TCP首部的固定长度。在以太网中使用IPv4的MSS值为1460，使用IPv6的MSS值为1440（两者的TCP首部都是20个字节，但IPv4首部是20字节，IPv6首部却是40字节）。在TCP的MSS选项中，MSS值是一个16位的字段，限定其最大值为65 535。这对于IPv4是适合的，因为IPv4数据报中的最大TCP数据量为65 495（65 535减去IPv4首部的20字节和TCP首部的20字节）。然而对于具有特大净荷选项的IPv6，却需要使用另外一种技巧（RFC 2675［Borman, Deering, and Hinden 1999］）。首先，没有特大净荷选项的IPv6数据报中的最大TCP数据量为65 515（65 535减去TCP首部的20字节）。65 535这个MSS值于是被视为表示“无限”的一个特殊值。该值只在用到特大净荷选项时才使用，不过这种情况却要求实际的MTU超过65 535。其次，如果TCP使用特大净荷选项，并且接收到的对端通告的MSS为65 535，那么它所发送数据报的大小限制就是接口MTU。如果这个值太大（也就是说所在路径中某个链路的MTU比较小），那么路径MTU发现功能将确定这个较小值。

SCTP基于到对端所有地址发现的最小路径MTU保持一个分片点。这个最小MTU大小用于把较大的用户消息分割成较小的能够以单个IP数据报发送的若干片段。SCTP_MAXSEG套接字选项可以影响该值，使得用户能够请求一个更小的分片点。

2.11.1 TCP输出

图2-15展示了某个应用进程写数据到一个TCP套接字中时发生的步骤。

每一个TCP套接字有一个发送缓冲区，我们可以使用SO_SNDBUF套接字选项来更改该缓冲区的大小（见7.5节）。当某个应用进程调用write时，内核从该应用进程的缓冲区中复制所有数据到所写套接字的发送缓冲区。如果该套接字的发送缓冲区容不下该应用进程的所有数据

（或是应用进程的缓冲区大于套接字的发送缓冲区，或是套接字的发送缓冲区中已有其他数据），该应用进程将被投入睡眠。这里假设该套接字是阻塞的，它是通常的默认设置。（我们将在第16章中阐述非阻塞的套接字。）内核将不从write系统调用返回，直到应用进程缓冲区中的所有数据都复制到套接字发送缓冲区。因此，从写一个TCP套接字的write调用成功返回仅仅表示我们可以重新使用原来的应用进程缓冲区，并不表明对端的TCP或应用进程已接收到数据。（我们将在7.5节随SO_LINGER套接字选项详细讨论这一点。）

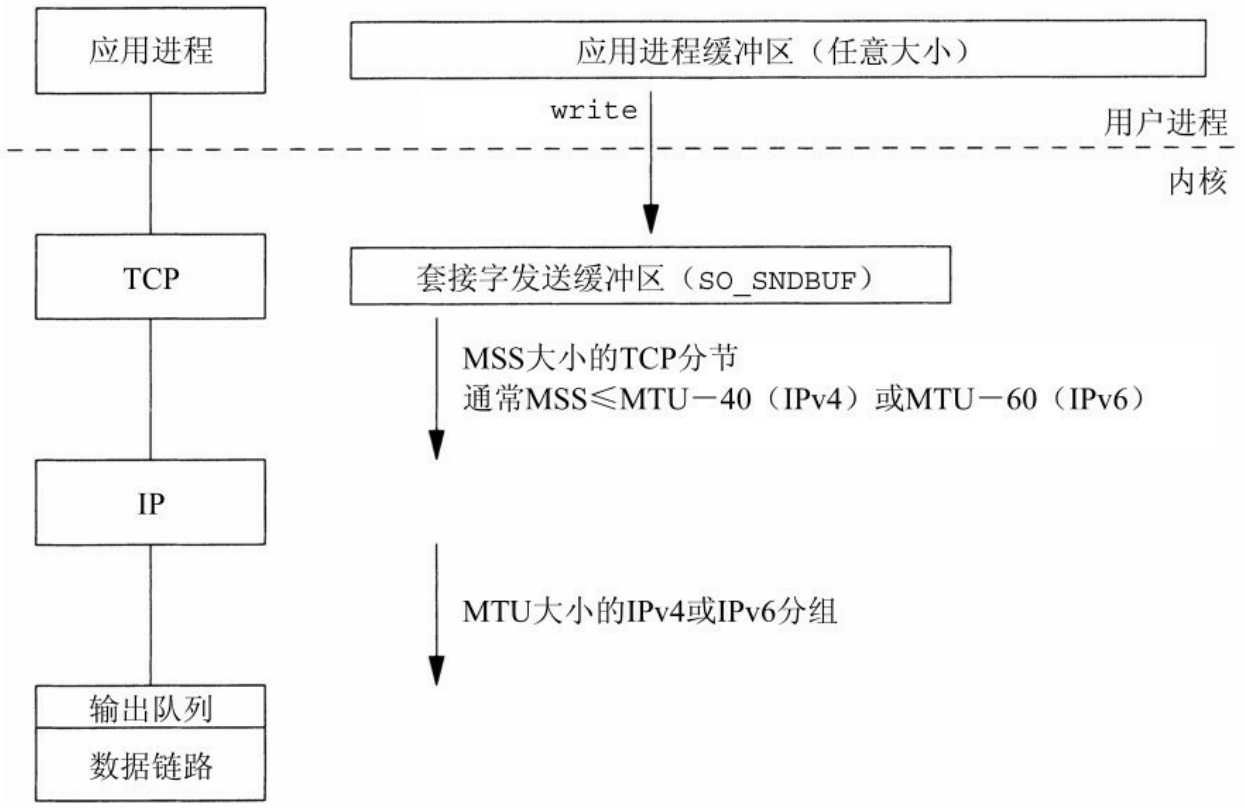


图2-15 应用进程写TCP套接字时涉及的步骤和缓冲区

这一端的TCP提取套接字发送缓冲区中的数据并把它发送给对端TCP，其过程基于TCP数据传送的所有规则（TCPv1的第19章和第20章）。对端TCP必须确认收到的数据，伴随来自对端的ACK的不断到达，本端TCP至此才能从套接字发送缓冲区中丢弃已确认的数据。TCP

必须为已发送的数据保留一个副本，直到它被对端确认为止。

本端TCP以MSS大小的或更小的块把数据传递给IP，同时给每个数据块安上一个TCP首部以构成TCP分节，其中MSS或是由对端通告的值，或是536（若对端未发送一个MSS选项）。（536是IPv4最小重组缓冲区字节数576减去IPv4首部字节数20和TCP首部字节数20的结果。）IP给每个TCP分节安上一个IP首部以构成IP数据报，并按照其目的IP地址查找路由表项以确定外出接口，然后把数据报传递给相应的数据链路。IP可能在把数据报传递给数据链路之前将其分片，不过我们已经谈到MSS选项的目的之一就是试图避免分片，较新的实现还使用了路径MTU发现功能。每个数据链路都有一个输出队列，如果该队列已满，那么新到的分组将被丢弃，并沿协议栈向上返回一个错误：从数据链路到IP，再从IP到TCP。TCP将注意到这个错误，并在以后某个时刻重传相应的分节。应用进程并不知道这种暂时的情况。

2.11.2 UDP输出

图2-16展示了某个应用进程写数据到一个UDP套接字中时发生的步骤。

这一次我们以虚线框展示套接字发送缓冲区，因为它实际上并不存在。任何UDP套接字都有发送缓冲区大小（我们可以使用SO_SNDBUF套接字选项更改它，见7.5节），不过它仅仅是可写到该套接字的UDP数据报的大小上限。如果一个应用进程写一个大于套接字发送缓冲区大小的数据报，内核将返回该进程一个EMSGSIZE错误。既然UDP是不可靠的，它不必保存应用进程数据的一个副本，因此无需一个真正的发送缓冲区。（应用进程的数据在沿协议栈向下传递时，通常被复制到某种格式的一个内核缓冲区中，然而当该数据被发送之后，这个副本就被数据链路层丢弃了。）

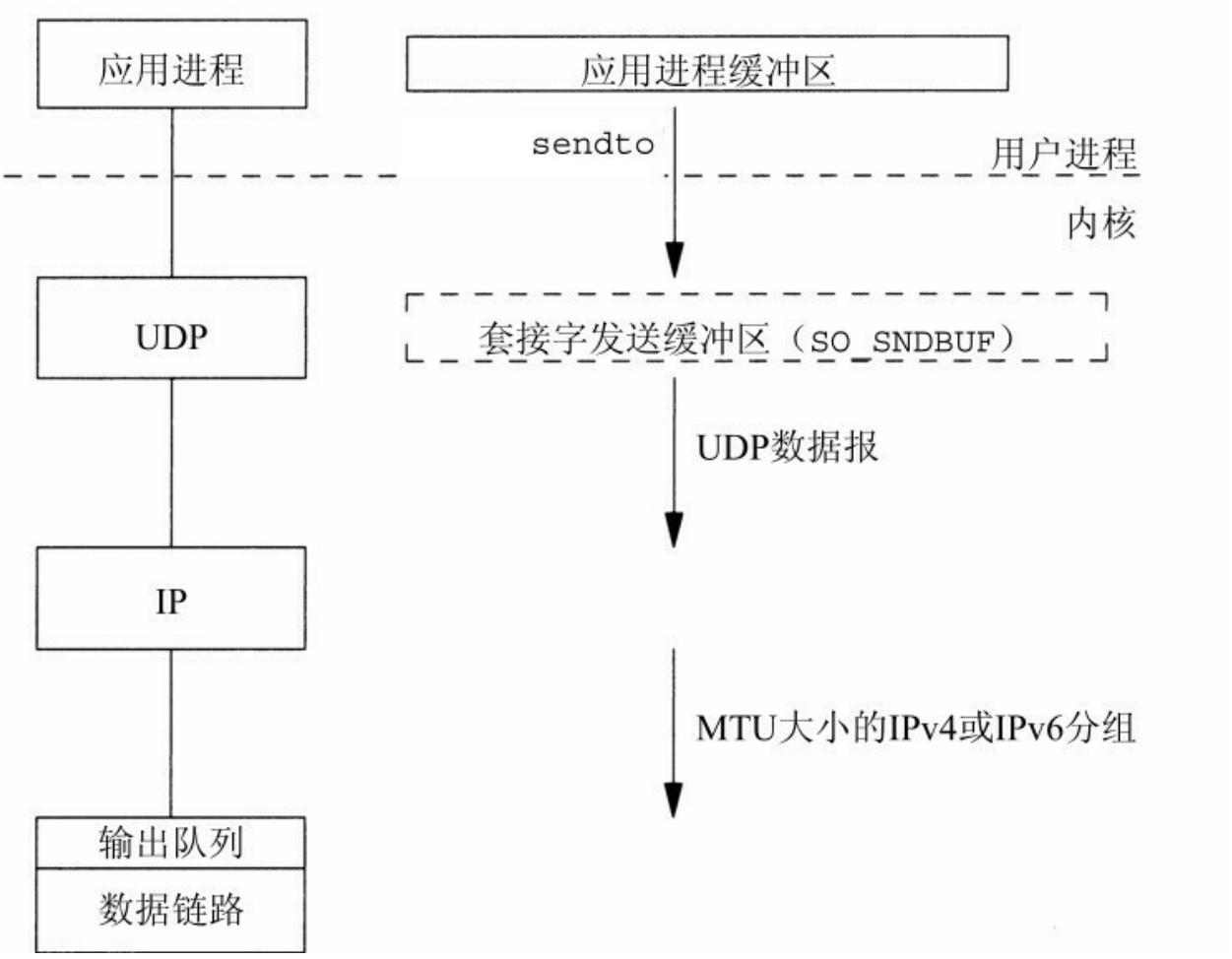


图2-16 应用进程写UDP套接字时涉及的步骤与缓冲区

这一端的UDP简单地给来自用户的数据报安上它的8字节的首部以构成UDP数据报，然后传递给IP。IPv4或IPv6给UDP数据报安上相应的IP首部以构成IP数据报，执行路由操作确定外出接口，然后或者直接把数据报加入数据链路层输出队列（如果适合于MTU），或者分片后再把每个片段加入数据链路层的输出队列。如果某个UDP应用进程发送大数据报（譬如说2000字节的数据报），那么它们相比TCP应用数据更有可能被分片，因为TCP会把应用数据划分成MSS大小的块，而UDP却没有对等的手段。

从写一个UDP套接字的write调用成功返回表示所写的数据报或其所有片段已被加入数据链路层的输出队列。如果该队列没有足够的空间存

放该数据报或它的某个片段，内核通常会返回一个ENOBUFS错误给它的应用进程。

不幸的是，有些UDP的实现不返回这种错误，这样甚至数据报未经发送就被丢弃的情况应用进程也不知道。

2.11.3 SCTP输出

图2-17展示了某个应用进程写数据到一个SCTP套接字中时发生的步骤。

既然SCTP是与TCP类似的可靠协议，它的套接字也有一个发送缓冲区，而且跟TCP一样，我们可以用SO_SNDBUF套接字选项来更改这个缓冲区的大小（见7.5节）。当一个应用进程调用write时，内核从该应用进程的缓冲区中复制所有数据到所写套接字的发送缓冲区。如果该套接字的发送缓冲区容不下该应用进程的所有数据（或是应用进程的缓冲区大于套接字的发送缓冲区，或是套接字的发送缓冲区中已有其他数据），应用进程将被投入睡眠。这里假设该套接字是阻塞的，它是通常的默认设置。（我们将在第16章中阐述非阻塞的套接字。）内核将不从write系统调用返回，直到应用进程缓冲区中的所有数据都复制到套接字发送缓冲区。因此，从写一个SCTP套接字的write调用成功返回仅仅表示我们可以重新使用原来的应用进程缓冲区，并不表明对端的SCTP或应用进程已接收到数据。

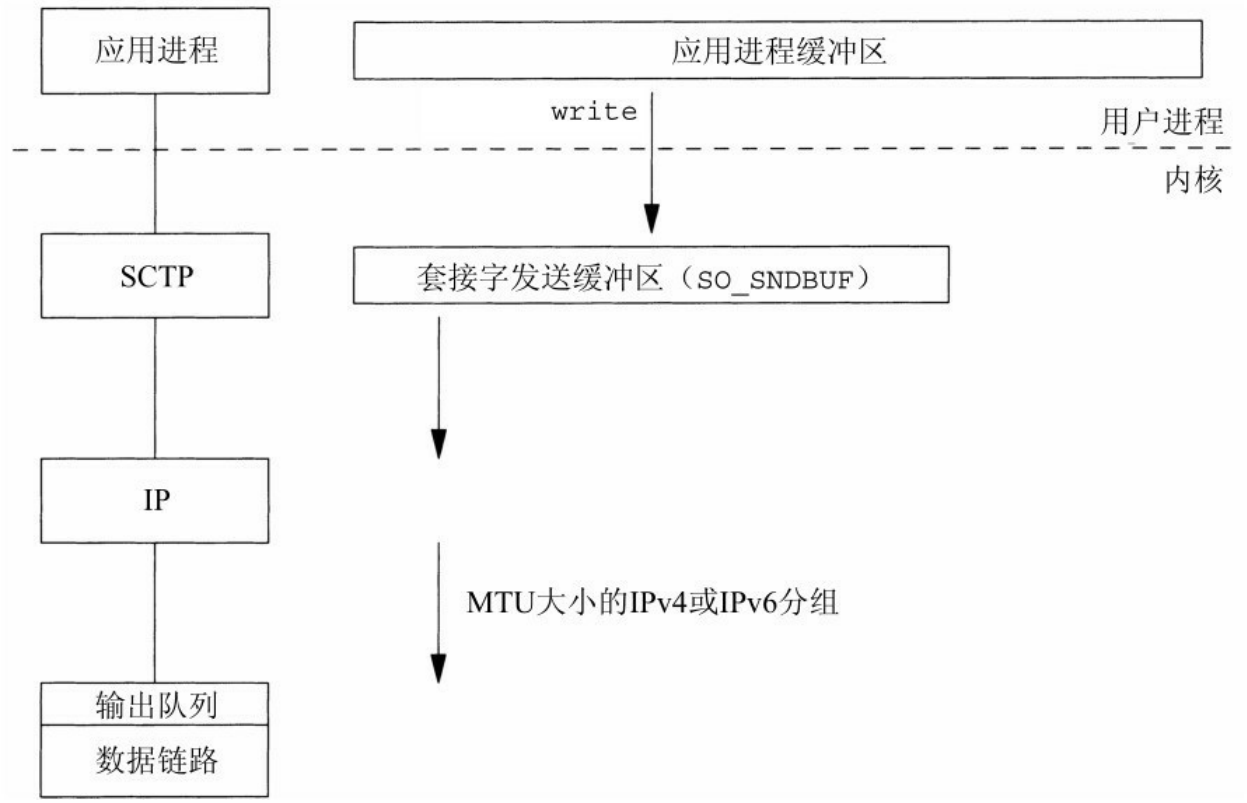


图2-17 应用进程写SCTP套接字时涉及的步骤和缓冲区

这一端的SCTP提取套接字发送缓冲区的数据并把它发送给对端SCTP，其过程基于SCTP数据传送的所有规则（数据传送的细节见［Stewart and Xie 2001］的第5章）。本端SCTP必须等待SACK，在累积确认点超过已发送的数据后，才可以从套接字缓冲区中删除该数据。

2.12 标准因特网服务

图2-18列出了TCP/IP多数实现都提供的若干标准服务。注意，表中所有服务同时使用TCP和UDP提供，并且这两个协议所用端口号也相同。

这些服务通常由Unix主机的inetd守护进程提供（见13.5节）。它们还提供使用标准的Telnet客户程序就能完成的简易测试机制。举例来说，下面就是时间获取和回射这两个标准服务器的测试过程：

```
aix % telnet freebsd daytime
Trying 12.106.32.254...                          Telnet客户输出
Connected to freebsd.unpbook.com.                Telnet客户输出
Escape character is '^]'.                        Telnet客户输出
Mon Jul 28 11:56:22 2003                         daytime服务器输出
Connection closed by foreign host.               Telnet客户输出（服务器关闭连接）
aix % telnet freebsd echo
Trying 12.106.32.254...                          Telnet客户输出
Connected to freebsd.unpbook.com.                Telnet客户输出
Escape character is '^]'.                        Telnet客户输出
hello,world                                      我们键入这行
hello,world                                      它由服务器回射回来
^]                                               键入Ctrl+]以与Telnet客户交谈
telnet>  quit                                    告诉客户我们已测试完毕
Connection  closed.                              这次客户自己关闭连接
```

名　　字	TCP端口	UDP端口	RFC	说　　明
echo（回射）	7	7	862	服务器返回客户发送的数据
discard（丢弃）	9	9	863	服务器废弃客户发送的数据
daytime（时间获取）	13	13	867	服务器返回直观可读的日期和时间
chargen（字符生成）	19	19	864	TCP服务器发送连续的字符流，直到客户终止连接。UDP服务器则每当客户发送一个数据报就返送一个包含随机数量（0~512）字符的数据报
time（流逝时间获取）	37	37	868	服务器返回一个32位二进制数值表示的时间。这个数值表示从1900年1月1日子时（UTC时间）以来所流逝的秒数

图2-18 大多数实现提供的标准TCP/IP服务 [12]

在这两个例子中，我们键入主机名和服务名（daytime和echo）。这些服务名由/etc/services文件映射到图2-18所示的端口号，详见11.5节。

注意，当我们连接到daytime服务器时，服务器执行主动关闭，然而当连接到echo服务器时，客户执行主动关闭。回顾图2-4，我们知道执行主动关闭的那一端就是历经TIME_WAIT状态的那一端。

为了应付针对它们的拒绝服务攻击和其他资源使用攻击，在如今的系统中，这些简单的服务通常被禁用。

2.13 常见因特网应用的协议使用

图2-19总结了各种常见的因特网应用对协议的使用情况。

前两个因特网应用ping和traceroute是使用ICMP协议实现的网络诊断应用。traceroute自行构造UDP分组来发送并读取所引发的ICMP应答。

紧接着是3个流行的路由协议，它们展示了路由协议使用的各种传输协议。OSPF通过原始套接字直接使用IP，RIP使用UDP，BGP使用TCP。

接下来5个是基于UDP的网络应用，然后是7个TCP网络应用和4个同时使用UDP和TCP的网络应用，最后5个是IP电话网络应用，它们或者独自使用SCTP，或者选用UDP、TCP或SCTP。

因特网应用	IP	ICMP	UDP	TCP	SCTP
ping		•			
traceroute		•	•		
OSPF（路由协议）	•				
RIP（路由协议）			•		
BGP（路由协议）				•	
BOOTP（引导协议）			•		
DHCP（引导协议）			•		
NTP（时间协议）			•		
TFTP（低级FTP）			•		
SNMP（网络管理）			•		
SMTP（电子邮件）				•	
Telnet（远程登录）				•	
SSH（安全的远程登录）				•	
FTP（文件传送）				•	
HTTP（Web）				•	
NNTP（网络新闻）				•	
LPR（远程打印）				•	
DNS（域名系统）			•	•	
NFS（网络文件系统）			•	•	
Sun RPC（远程过程调用）			•	•	
DCE RPC（远程过程调用）			•	•	
IUA（IP之上的ISDN）					•
M2UA/M3UA（SS7电话信令）					•
H.248（媒体网关控制）			•	•	•
H.323（IP电话）			•	•	•
SIP（IP电话）			•	•	•

图2-19 各种常见因特网应用的协议使用情况

2.14 小结

*